Is Exceptional Behavior Testing an Exception? An Empirical Assessment Using Java Automated Tests

Francisco Dalton
Federal University of Alagoas (UFAL)
Maceió, Alagoas, Brazil
fdbd@ic.ufal.br

Márcio Ribeiro
Federal University of Alagoas (UFAL)
Maceió, Alagoas, Brazil
marcio@ic.ufal.br

Gustavo Pinto
Federal University of Pará (UFPA)
Belém, Pará, Brazil
gpinto@upfa.br

Leo Fernandes
Federal Institute of Alagoas (IFAL)
Maceió, Alagoas, Brazil
leonardo.oliveira@ifal.edu.br

Rohit Gheyi
Federal University of Campina Grande (UFCG)
Campina Grande, Paraíba, Brazil
rohit@dsc.ufcg.edu.br

Baldoino Fonseca
Federal University of Alagoas (UFAL)
Maceió, Alagoas, Brazil
baldoino@ic.ufal.br

ABSTRACT

Software testing is a crucial activity to check the internal quality of a software. During testing, developers often create tests for the normal behavior of a particular functionality (e.g., was this file properly uploaded to the cloud?). However, little is known whether developers also create tests for the exceptional behavior (e.g., what happens if the network fails during the file upload?). To minimize this knowledge gap, in this paper we design and perform a mixed-method study to understand how 417 open source Java projects are testing the exceptional behavior using the JUnit and TestNG frameworks, and the AssertJ library. We found that 254 (60.91%) projects have at least one test method dedicated to test the exceptional behavior. We also found that the number of test methods for exceptional behavior with respect to the total number of test methods lies between 0% and 10% in 317 (76.02%) projects. Also, 239 (57.31%) projects test only up to 10% of the used exceptions in the System Under Test (SUT). When it comes to mobile apps, we found that, in general, developers pay less attention to exceptional behavior tests when compared to desktop/server and multi-platform developers. In general, we found more test methods covering custom exceptions (the ones created in the own project) when compared to standard exceptions available in the Java Development Kit (JDK) or in third-party libraries. To triangulate the results, we conduct a survey with 66 developers from the projects we study. In general, the survey results confirm our findings. In particular, the majority of the respondents agrees that developers often neglect exceptional behavior tests. As implications, our numbers might be important to alert developers that more effort should be placed on creating tests for the exceptional behavior.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Exceptions, Exceptional Behavior, Software Testing.

ACM Reference Format:
Francisco Dalton, Márcio Ribeiro, Gustavo Pinto, Leo Fernandes, Rohit Gheyi, and Baldoino Fonseca. 2020. Is Exceptional Behavior Testing an Exception? An Empirical Assessment Using Java Automated Tests. In *Evaluation and Assessment in Software Engineering (EASE 2020), April 15–17, 2020, Trondheim, Norway.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3383219.3383237

1 INTRODUCTION

Exception handling techniques are important in modern object-oriented software development. With exceptions, it is possible to provide greater reliability in the systems' execution flow, as they allow abnormal behavior to be detected, reported, handled, and corrected, when possible [2, 6, 26]. Hence, there are several studies that try to assess the quality of exception handling code [4, 6], development patterns [7, 19, 21, 25], or best practices and usage scenarios [5, 8, 16, 22]. These studies helped researchers and practitioners to better understand and shape novel exception handling constructs, techniques, and tools.

In this context, exceptional behavior scenarios should be tested in order to guarantee that an eventual anomalous behavior will be detected or handled accordingly. Unfortunately, previous studies have provided initial evidences—based on a study with 10 projects—that software developers tend to neglect exception behavior testing [3, 13]. This finding is particularly worrying, since the absence of tests aimed to validate the launching and handling of exceptions can compromise precisely their core feature: the reliability expected to be obtained from their use [7, 8, 17]. In fact, studies provide evidences that the majority of crashes in Android Apps are related to exceptions defined in the Android Framework [11]. Thus, it is possible that a software system presents failures that could be otherwise avoided through more rigorous testing that handle exceptional behavior [3, 14], an activity that we call throughout this paper as "exceptional behavior testing."

Some natural questions that one may raise in this context are: How common is for developers to test the exceptional behavior? Are these tests more common in desktop/server projects when compared to mobile projects? Do developers prioritize testing custom exceptions (the ones created in the own project) or standard/third-party exceptions (from the Java development kit and third-party libraries)? Unfortunately, despite the vast number of studies that dealt with exception-handling constructs [5, 8, 16, 21, 22, 25], the literature is not particularly rich when it comes to empirical studies that shed evidence on whether developers create tests for the exceptional behavior of their software systems.

To better understand the landscape of exceptional behavior testing in practice, in this paper we present a *mixed-method* study considering: (1) an empirical investigation of over 346,573 test methods from 417 open source Java projects and (2) a survey with 66 developers from these projects. We employed several criteria for selecting our corpus of projects, such as the use of JUnit,[1] TestNG,[2] or AssertJ,[3] and the use of exceptions. We sorted these projects by popularity, measured in terms of the number of stars (as of October 2019). For each project, we selected and downloaded the latest version available. We categorize these projects using two dimensions: the platforms (i.e., desktop/server, mobile, or multi-platform) and the domains (i.e., framework, library, or tool). We then created a tool that collects metrics related to exception-handling constructs [9] (i.e., `throw` statements, `throws` clauses, and `catch` blocks) in the System Under Test (SUT), and exceptions definitions (custom or standard/third-party). Also, we collect metrics related to exception-testing constructs (e.g., the `expected` attribute of the `@Test` annotation, `fail` call right before a `catch` block) of the JUnit and TestNG frameworks, and the AssertJ library, among many other metrics.

Our results indicate that the majority of the studied projects—254 out of 417 (60.91%)—has at least one test method to deal with the exceptional behavior. However, we found that the number of test methods for exceptional behavior with respect to the total number of test methods lies between 0% and 10% in 317 (76.02%) projects. Also, 239 (57.31%) projects test only up to 10% of the used exceptions in the SUT. Moreover, we found that mobile developers tend to create less exceptional behavior test methods than developers of the other two platforms. When considering the projects' domains, libraries developers create more exceptional behavior test methods than developers of the other domains. We also observed that developers often create more exceptional behavior test methods that cover custom exceptions in 55.51% of the projects than standard/third party exceptions. This is particularly the case of desktop/server (65.38%) and multi-platform (61.74%) projects. Regarding the survey, the answers in general are in accordance to our results. The majority of the respondents (69.70%) believe that exceptional behavior testing is important. However, 53% agrees that developers often neglect such tests. Our respondents also noticed that they prioritize the creation of tests that focus on custom exceptions over standard/third-party ones.

The main contributions of this work is three-fold:

- a quantitative study (based on 417 open source projects) to understand whether and how developers test the exceptional behavior of their software systems;
- a survey with 66 developers (from the projects we studied) to triangulate with our quantitative results; and
- a tool that is able to, given a Git repository, report a set of metrics related to testing exceptional behavior.

[1]https://junit.org/junit5/
[2]https://testng.org/doc/
[3]https://assertj.github.io/

In this paper, we have the following scope: we focus on Java automated tests written using JUnit, TestNG, or AssertJ; our tool collects the metrics statically, i.e., we do not run the programs; we collect all the exceptions used in the SUT (checked and unchecked). All projects we use in this study are open source.

2 MOTIVATING SCENARIO

We refer to "exceptional behavior testing" as test methods that expect exceptions to be raised. We illustrate an example in Listing 1. The test passes in case `IllegalArgumentException` is raised.

```
1  @Test(expected = IllegalArgumentException.class)
2  public void negative_throws(){
3      new TakeIterable<>(Interval.oneTo(5), -1);
4  }
```

Listing 1: Exceptional behavior test method example.

Many studies on exceptions and error handling have been developed along the years [3, 13, 27]. In this context, although the goal of these works is not to analyze whether or how developers test the exceptional behavior, they suggest that testing the exceptional behavior is not quite common. For instance, Goffi *et al.* [13] claimed that developers "*do not pay equal attention to testing exceptional behavior.*" Similarly, Bernardo *et al.* [3] claimed that "*manually-written test suites tend to neglect exceptional behavior.*"

Listing 1 was extracted from the eclipse-collections project.[4] To evaluate the exceptional behavior, this particular test method relies on the `expected` attribute of the `@Test` annotation (Line 2) provided by JUnit. This project has 1,726 out of 10,362 test methods (16.66%) that evaluate the exceptional behavior. However, there are projects that place less effort in evaluating the exceptional behavior. For example, despite the similar number of test methods, the ghidra framework[5] (a software reverse engineering framework created and maintained by the National Security Agency) has 348 out of 10,976 test methods (3.17%) aimed to evaluate the exceptional behavior. Given this scenario, although existing works [3, 13] claim that developers neglect exceptional behavior testing, they did not provide an in-depth investigation on whether and how developers test the exceptional behavior. In this work, we consider a much higher number of projects (417 *versus* 10 [13]). We also identify differences among platforms and domains, correlate the results with repositories' characteristics, and triangulate our results with a survey with 66 participants.

3 EMPIRICAL STUDY

In this section we present our empirical study. First, we introduce the research questions (Section 3.1). Then we present the studied projects and the criteria used to select them (Section 3.2). Afterwards, we detail the metrics we use to answer our research questions (Section 3.3). Also, we describe our tool used to collect and

[4]https://github.com/eclipse/eclipse-collections
[5]https://github.com/NationalSecurityAgency/ghidra

analyze data (Section 3.4). Finally, we explain the procedures we use to perform our survey (Section 3.5).

3.1 Goal and Research Questions

The goal of our study consists of statically analyzing open source projects for the purpose of assessing whether and how developers create exceptional behavior test methods from the point of view of software developers in the context of open source projects.

We intend to answer the following research questions:

- **RQ1:** Do developers create test methods for the exceptional behavior?
- **RQ2:** Do the test methods cover more distinct custom exceptions or distinct standard/third-party exceptions?
- **RQ3:** How do developers perceive the exceptional behavior testing?

Answering **RQ1** is important to understand if developers intentionally create exceptional behavior tests for the exceptions found in the SUT, and if it is possible to notice different results when comparing distinct software platforms and domains. Answering **RQ2** is important to verify if developers pay equal attention to the exceptional behavior regardless of the source of the exception (i.e., custom or standard/third-party). Answering **RQ3** is important to comprehend how developers perceive the exceptional behavior testing and to raise developers thoughts and opinions. This might help researchers and practitioners with developing processes and tools to focus on exceptional behavior tests.

3.2 Studied Projects

To select the projects for our study, we used the GitHub API to query and find repositories. We focused on frameworks, libraries, and tools written in Java. Then, we sorted the resulting list of projects by the number of stars. As an example, to find libraries we executed the following query: `language: java sort: stars library`. As a stop criteria, we arbitrarily limited our script to fetch 600 repositories. However, some of these projects do not exhibit the characteristics we are interested. We then excluded repositories that do not meet the following criteria:

(1) Has at least one custom, standard, or third-party exception being used in exception-handling constructs (i.e., `throws` clauses, `throw` statements, or on `catch` blocks) in the SUT;
(2) Has at least one test method using JUnit, TestNG, or AssertJ.

Criterion (1) indicates that the project under evaluation makes use of exceptions and, therefore, developers may have a reason to implement tests for exceptional behavior. Criterion (2) was designed to eliminate projects that do not do any automated testing using JUnit, TestNG, or AssertJ. Criterion (1) excluded 162 projects while Criterion (2) excluded 21 more projects. Thus, the empirical study we report in this paper considers 417 projects.

In the next step, we classified each project considering the platform. In particular, we focused on desktop/server (exclusively), mobile (exclusively), and multi-platform. To perform this classification, we rely on the javalibs.com website. Given a project, this website returns—based on maven dependencies—whether the project is used by mobile and non-mobile projects. For example, when considering the RxJava project, the website reports that 96% of the projects that use RxJava are non-Android projects and 4% of the projects that use RxJava are Android projects. Therefore, we classify RxJava as multi-platform. Afterwards, two researchers manually analyzed each project to confirm the website classification. For the projects that are not available in the website, we rely exclusively on our manual classification.

Some of the projects we use in our empirical study include dropwizard, antlr4, eclipse-collections, docx4j, netty (classified as desktop/server); bento, hover, picasso, zxing-android-embedded, joda-time-android, tinker (classified as mobile); and selenium, jacoco, guava, junit4, google-cloud-java, google-maps-services-java, mockito, soot, spring-boot, spring-framework (classified as multi-platform). In summary, our dataset has 202 (48.44%) desktop/server projects (78 frameworks, 34 libraries, and 90 tools); 152 (36.45%) mobile projects (50 frameworks, 60 libraries, and 42 tools); and 63 (15.11%) multi-platform projects (22 frameworks, 35 libraries, and six tools). Figure 1 illustrates distributions regarding the repositories ages, repository activity, LOC, stars, and contributors of the projects.

3.3 Collected Metrics

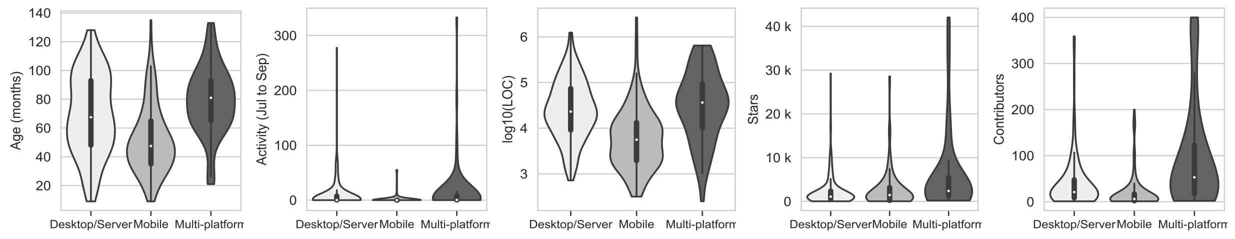
(1) *Number of Distinct Used Exceptions **(NDUE)***: Number of distinct exceptions found in `throw` instructions, `throws` clauses, and `catch` blocks in the SUT;
(2) *Number of Distinct Used Custom Exceptions **(NDUCE)***: Number of distinct custom exceptions found in `throw` instructions, `throws` clauses, and `catch` blocks in the SUT;
(3) *Number of Distinct Used Standard/Third-party Exceptions **(NDUSTE)***: Number of distinct standard/third-party exceptions found in `throw` instructions, `throws` clauses, and `catch` blocks in the SUT;
(4) *Number of Test Methods **(NTM)***: Total number of test methods;
(5) *Number of Exceptional Behavior Test Methods **(NEBTM)***: Total number of test methods with exception-testing constructs;
(6) *Number of Distinct Tested Exceptions **(NDTE)***: Total number of distinct exceptions used in at least one test method and in the SUT;
(7) *Number of Distinct Tested Custom Exceptions **(NDTCE)***: Total number of distinct custom exceptions used in at least one test method and in the SUT;
(8) *Number of Distinct Tested Standard/Third-party Exceptions **(NDTSTE)***: Total number of distinct standard/third-party exceptions used in at least one test method and in the SUT.

3.4 Tool Description and Usage Scenarios

We developed a tool [10] that analyzes Git repositories. For each repository, the tool performs a syntactic analysis on Java code. This tool was based on JavaParser,[6] a lightweight library for supporting syntactic analysis in Java code, version 3.13.2.

Our tool identifies exception-handling constructs [9] (i.e., `throw`, `throws`, and `catch`) in the SUT. It also identifies exception-testing constructs from JUnit (i.e., `assertThrows` call, the `expected` attribute of the `@Test` annotation, and the `ExpectedException` rule), from TestNG (i.e., the `expectedExceptions` attribute of the `@Test`

[6]https://javaparser.org



(a) Ages. **(b) Commits per month.** **(c) LOC.** **(d) Stars.** **(e) Contributors.**

Figure 1: Dataset distributions. The white dot represents the median.

```
1 public <T> T convertIfNecessary(...) throws
      TypeMismatchException {
2    try {
3       return this.typeConverterDelegate...;
4    } catch (ConverterNotFoundException | IllegalStateException
          ex) {
5       throw new ConversionNotSupportedException(value,
             requiredType, ex);
6    } catch (ConversionException | IllegalArgumentException ex) {
7       throw new TypeMismatchException(value, requiredType, ex);
8    }
9 }
```

Listing 2: Examples of exception-handling constructs.

annotation), and from AssertJ library calls (i.e., `assertThatThrownBy`, `asserThatExceptionOfType`, `assertThatIOException`). Also, our tool identifies a `fail` call right before a `catch` block, which is common in tests written in JUnit, TestNG, and AssertJ. During the identification of the constructs, our tool collects the exceptions' names being used in the source code and in the test methods.

To better explain how our tool works, consider the code snippet from the Spring-Framework[7] project shown in Listing 2. If this code was analyzed by our tool, the exception `TypeMismatchException` described in the `throws` clause (Line 2) would be collected. Moreover, the exceptions `ConvertedNotFoundException` (Line 6), `IllegalStateException` (Line 6), `ConversionException` (Line 9), and `IllegalArgumentException` (Line 9) would also be collected, because they are used inside `catch` blocks. Finally, the exceptions `ConversionNotSupportedException` (Line 7) and `TypeMismatchException` (Line 10) would also be collected because they are used within a `throw` instruction.

Listing 3 shows a test method to test exceptional behavior. In this case, our tool would collect the exception in the first parameter of the `assertThrows` method (Line 4), which in this case is the `IllegalStateException` exception.

```
1 @Test
2 void invalidExpressionEvaluationType() {
3    IllegalStateException exception =
          assertThrows(IllegalStateException.class, ...);
4 }
```

Listing 3: `assertThrows` method call example.

Similarly, we can collect the exceptions used in the `expected` attribute of the `@Test` annotations (see Listing 1, Line 2). Our tool can also collect the exceptions used with the `ExpectedException` rule. Listing 4 illustrates an example. In this case, the tool would collect the `IllegalArgumentException` exception (Line 6).

[7]https://github.com/spring-projects/spring-framework

Table 1: Three metrics results from our code snippets.

Metric	Custom	Standard or Third-Party
NDUE	4	2
NDTE	1	2
NEBTM	1	2

```
1 @Rule
2 public ExpectedException expectedException =
       ExpectedException.none();
3 @Test
4 public void addPropertiesFilesToEnvironmentWithNullContext() {
5    expectedException.expect(IllegalArgumentException.class);
6 }
```

Listing 4: `ExpectedException` rule example.

Our tool also collects the exceptions being used in the `catch` blocks right after a call to the `fail` method (Listing 5). To test the exceptional behavior, the developer expects the `TypeMismatchException` (Line 6) to be thrown and thus the test execution would not reach the `fail` method call. In case the execution reaches the `fail` method call, the test fails.

```
1 @Test
2 public void setEnumProperty() {
3    try {
4       fail("Should have thrown TypeMismatchException");
5    } catch (TypeMismatchException ex) {
6       (...)
7    }
8 }
```

Listing 5: `fail` method call example.

The tool also labels each exception as custom (an exception created in the own project), standard (readily available in the Java development kit), or third-party (available in third-party libraries exceptions).

Table 1 shows the exceptions found in Listings 2–5. The `IllegalArgumentException`, and `IllegalStateException` exceptions are labelled as standard/third-party. The remaining ones are labelled as custom exceptions. In this case, we can see that our tool would have found six distinct exceptions being used in the SUT and would have found tests only for three exceptions. This tell us that 50% of the distinct used exceptions have been tested.

3.5 Survey Data

Our survey intends to cross-validate the findings observed in the mining study. The survey has two main sections. The first focuses on developers background and general information (i.e., Java experience, platforms, and domains in which they work with, and demographics information). The second section asks participants (i) to rate the importance of exceptional behavior testing, (ii) whether they prioritize custom or standard/third-party exceptions when writing tests, and (iii) to evaluate the following sentence: "*Software developers neglect tests that focus on exceptional behavior.*"

The participants of our survey are developers of the projects we studied. We developed a script that selects the contributors that made commits with the "`test`" keyword in the commit message. We sent e-mails to 2,259 developers inviting them to participate in the survey. We sent the actual questionnaire on October 8th, 2019. During the period of 14 days we received 66 responses (a 2.92% response ratio). The respondents are from North America (22.73%), South America (3.03%), Europe (60.61%), Asia (6.06%), Eurasia (4.55%), Oceania (1.52%), and undefined (1.52%). The majority of the participants (60.60%) has more than 10 years of experience in Java. The respondents are knowledgeable (63.60%) and very knowledgeable (33.40%) with Java Testing frameworks.

4 RESULTS AND DISCUSSION

In this section we answer our research questions and discuss the results we obtained. All data, scripts and the tool created in this study are also online available in our companion website [10].

RQ1: Do developers create test methods for the exceptional behavior?

We found that 254 out of 417 (60.91%) projects have at least one test method for exceptional behavior (NEBTM > 0). When considering the platforms, we notice that 149 out of 202 (73.76%) projects of the desktop/server platform and 52 out of 63 (82.54%) multi-platform projects have NEBTM above zero. The result is much lower when considering the mobile platform: only 53 out of 152 (34.87%) projects have at least one test method for exceptional behavior. Regarding the domain, libraries have the highest numbers in all platforms: 85.29% (desktop/server), 41.67% (mobile), and 82.86% (multi-platform) of the libraries have at least one test method for exceptional behavior.

However, the numbers we illustrated so far do not have a ratio related to the total number of test methods of each project. To do so, for each project, we divided the number of exceptional behavior test methods by the total number of test methods (NEBTM/NTM in Table 2). As illustrated in Figure 2 (at the right-hand side, combining all domains), the great majority of the projects—317 out of 417 (76.02%)—dedicate up to 10% of the test methods to exceptional behavior. More specifically, the number of test methods for exceptional behavior with respect to the total number of test methods lies between 0% and 10% in 76.24% of the desktop/server projects; in 84.87% of the mobile projects; and in 53.97% of the multi-platform projects. The medians for NEBTM/NTM in Table 2 are: 4.02% for desktop/server; 0% for mobile; and 9.38% for multi-platform.

We also found strong Spearman [1] correlations in all platforms regarding NEBTM and non-Exceptional Behavior Test Methods (NTM-NEBTM), i.e., 0.84 (p-value 2.61^{-55}) for desktop/server, 0.76 (p-value 6.88^{-30}) for mobile, and 0.84 (p-value 1.72^{-18}) for multi-platform. This way, the lack of exceptional behavior test methods might be related to the absence of test methods in general.

Table 2: Collected metrics to answer RQ1. ***NEBTM = Number of Exceptional Behavior Test Methods; NTM = Number of Test Methods; NDUE = Number of Distinct Used Exceptions; NDTE = Number of Distinct Tested Exceptions.***

		NEBTM	NTM	NEBTM/ NTM	NDUE	NDTE	NDTE/ NDUE	
Desktop/Server	Frameworks	119.15	1,122.59	7.27%	50.92	10.64	18.44%	**mean**
		330.35	2,624.11	12.33%	46.68	16	19.49%	**std**
		0	1	0%	3	0	0%	**min**
		9.50	256	4%	37	4	12.13%	**50%**
		2,111	16,699	100%	252	77	73.91%	**max**
	Libraries	107.47	1,119.50	10.31%	56.21	13.38	26.45%	**mean**
		264.75	3,309.40	10.63%	78.92	20.20	24.44%	**std**
		0	15	0%	1	0	0%	**min**
		21	362	8.42%	34.50	7	20.94%	**50%**
		1,519	19,556	40%	460	104	100%	**max**
	Tools	60.69	617.33	7.80%	38.79	5.42	11.46%	**mean**
		241.42	1,814.61	14.12%	38.89	13.69	14.48%	**std**
		0	1	0%	3	0	0%	**min**
		4	97	3.06%	27.50	2	5.38%	**50%**
		2,164	15,707	84.73%	257	118	66.67%	**max**
	All	91.14	896.96	8.02%	46.41	8.78	16.68%	**mean**
		282.64	2,440.73	12.89%	50.77	16.06	19.13%	**std**
		0	1	0%	1	0	0%	**min**
		7	194.50	4.02%	34	3	10.20%	**50%**
		2,164	19,556	100%	460	118	100%	**max**
Mobile	Frameworks	39.70	655.10	2.89%	31.56	4.90	12.48%	**mean**
		134.05	2,345.87	4.80%	55.89	12.30	23.71%	**std**
		0	1	0%	1	0	0%	**min**
		0	16.50	0%	12	0	0%	**50%**
		806	15,077	16.86%	337	65	100%	**max**
	Libraries	11.22	134.98	4.29%	14.18	2.02	12.68%	**mean**
		28.24	330.57	7.02%	19.31	4.09	20.12%	**std**
		0	1	0%	1	0	0%	**min**
		0	22.50	0%	9	0	0%	**50%**
		144	1,922	26.26%	130	17	100%	**max**
	Tools	1.69	24.26	4.30%	16.50	0.43	2.87%	**mean**
		5.09	50.13	11.12%	14.86	1.11	8.12%	**std**
		0	1	0%	1	0	0%	**min**
		0	4.50	0%	12	0	0%	**50%**
		28	276	48.15%	60	6	40%	**max**
	All	17.95	275.48	3.84%	20.54	2.53	9.90%	**mean**
		79.99	1,379.23	7.79%	35.78	7.69	19.43%	**std**
		0	1	0%	1	0	0%	**min**
		0	10	0%	10	0	0%	**50%**
		806	15,077	48.15%	337	65	100%	**max**
Multi-platform	Frameworks	362.50	2,387.05	13.21%	68.18	28.41	37.41%	**mean**
		598.07	3,933.30	11.11%	78.92	44.06	37.64%	**std**
		0	2	0%	4	0	0%	**min**
		79.50	1,163	11.04%	46	12	34.20%	**50%**
		2,329	16,924	34.38%	374	194	160.53%	**max**
	Libraries	259.03	1,936.46	11.32%	33.43	12.03	32.58%	**mean**
		661.27	4,430.93	11.32%	28.30	14.96	32.82%	**std**
		0	6	0%	2	0	0%	**min**
		35	391	9.65%	26	8	21.43%	**50%**
		3,462	22,539	41%	122	70	125%	**max**
	Tools	20.17	537.33	3.67%	39	3.83	10.82%	**mean**
		41.16	614.41	3.75%	37.68	6.05	12.81%	**std**
		0	39	0%	11	0	0%	**min**
		4	235	2.86%	27.50	1.50	5.56%	**50%**
		104	1,333	7.97%	108	16	33.33%	**max**
	All	272.41	1,960.56	11.25%	46.10	16.97	32.19%	**mean**
		608.46	4,037.08	10.96%	54.14	29.33	33.73%	**std**
		0	2	0%	2	0	0%	**min**
		40	409	9.38%	36	7	20%	**50%**
		3,462	22,539	41%	374	194	160.53%	**max**

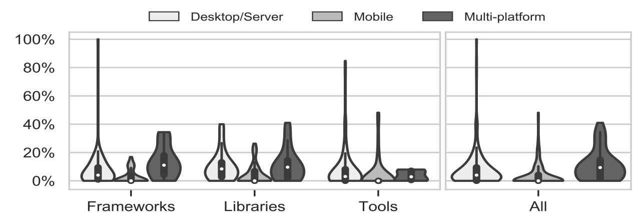


Figure 2: Ratio of NEBTM/NTM.

Furthermore, for each project we also calculate the ratio of the Number of Distinct Tested Exceptions (NDTUE) to the Number of Distinct Used Exceptions (NDUE) (see column NDTE/NDUE in Table 2). Our intention is to understand whether each of the exceptions used in `throw` instructions, `throws` clauses, and `catch` blocks has at least one corresponding test method. Figure 3 illustrates the distribution of these ratios for all projects. We notice that 239 out of 417 (57.31%) projects test only up to 10% of the used exceptions. When considering the platform, 52.97% (desktop/server), 73.03% (mobile), and 33.33% (multi-platform) of the projects test up to 10% of the used exceptions. In terms of domain, 55.33% (frameworks), 42.64% (libraries), and 73.19% (tools) of the projects test up to 10% of the used exceptions. We found medium Spearman correlations in all platforms regarding NDTUE/NDUE and the number of contributors of the projects, i.e., 0.37 (p-value 4.79^{-8}) for desktop/server, 0.42 (p-value 4.23^{-8}) for mobile, and 0.51 (p-value 1.83^{-5}) for multi-platform. This way, as the number of contributors grows, there might be a better chance of also growing the number of test methods for exceptions used in the SUT.

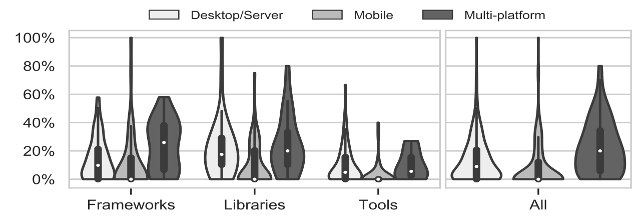


Figure 3: Ratio of NDTE/NDUE.

We observed that libraries developers tend to write more exceptional behavior test methods to a higher number of distinct exceptions than developers of the other two domains. In contrast, developers of the mobile platform tend to write less exceptional behavior test methods when compared to developers of the other two platforms. Previous work has reported a large study with 2,486 open-source Android apps and found that the majority of the crashes in these apps is related to exceptions defined in the Android framework [11]. Since the great majority of the mobile projects studied are from the Android platform, an interesting hypothesis is to test whether the lack of exceptional behavior tests is leading to the reported crashes. We can also check if the maturity of the majority of the multi-platform projects (e.g., Spring-framework, jacoco, junit4, mockito, RxJava, and selenium) is leading to better numbers than the desktop/server and mobile platforms. However, testing these hypotheses is out of the scope of this paper.

We also observed several projects (127 out of 417, i.e., 30.45%) creating test methods for exceptions not used in the SUT. This means that there is, for example, a test method with `@Test(expected = E.class)` and `E` is not used in the SUT. For example, the mockito project has test methods for 53 exceptions, but only 36 of them are used in the SUT.

RQ2: Do the test methods cover more distinct custom exceptions or distinct standard/third-party exceptions?

Table 3 presents the summary of the Number of Distinct Used Custom Exceptions (NDUCE) and the Number of Distinct Used Standard/Third-party Exceptions (NDUSTE) of all projects. Column NDUE represents the Number of Distinct Used Exceptions and is calculated by the sum of NDUCE and NDUSTE. Notice that the sum of both median ratios (NDUCE/NDUE + NDUSTE/NDUE) is 100% (see the median values of NDUCE/NDUE and NDUSTE/NDUE in Table 3, e.g., 10% and 90% in multi-platform libraries, respectively). According to the results, developers tend to use more commonly standard/third-party exceptions than to create and use new ones in their projects. Figure 4 presents the distributions of NDUCE/NDUE and NDUSTE/NDUE. Notice that NDUSTE/NDUE has higher ratios. Also, notice that the NDUCE/NDUE ratio is generally below 40%. The highest median ratio is achieved by the multi-platform frameworks (23.53%). The opposite (NDUCE/NDUE > NDUSTE/NDUE) happens in only seven projects: ghidra, XChange, j2objc, cosbench, spring-framework, airline, and platform_frameworks_base. In addition, the multi-platform tools are the only ones to use custom exceptions in all projects, but the sampling of this group is very small (i.e., six projects), as presented in the row "projects" in Table 3.

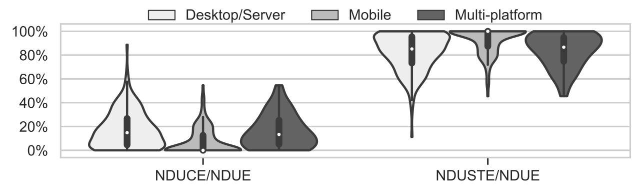


Figure 4: Ratios of NDUCE/NDUE and NDUSTE/NDUE.

Table 3 also presents the Number of Distinct Tested Custom Exceptions (NDTCE) and the Number of Distinct Tested Standard/Third-party Exceptions (NDTSTE). To analyze the distribution of these metrics related to the Number of Distinct Tested Exceptions (NDTE), for all projects we also compute NDTCE/NDTE and NDTSTE/NDTE (again, the sum of both ratios is 100%). However, we have projects where no exceptions have associated tests. Thus, we removed these projects to avoid divisions by zero. Therefore, the row "projects" in Table 3 has different numbers (e.g., in the first row (desktop/server frameworks), the number of projects dropped from 78 to 56 projects). Figure 5 shows the distribution of the NDTCE/NDTE and NDTSTE/NDTE ratios. Notice that in the majority of the projects there are more distinct standard/third-party exceptions with test methods when compared to distinct custom exceptions.

Our results suggest that standard/third party exceptions are more used throughout the SUT (i.e., ratios NDUCE/NDUE and

Table 3: Collected metrics to answer RQ2. *NDUCE = Number of Distinct Used Custom Exceptions; NDUSTE = Number of Distinct Used Standard/Third-party Exceptions; NDUE = Number of Distinct Used Exceptions; NDTCE = Number of Distinct Tested Custom Exceptions; NDTSTE = Number of Distinct Tested Standard/Third-party Exceptions; NDTE = Number of Distinct Tested Exceptions.*

		NDUCE	NDUSTE	NDUE	NDUCE/ NDUE	NDUSTE/ NDUE	NDTCE	NDTSTE	NDTE	NDTCE/ NDTE	NDTSTE/ NDTE	NDTCE/ NDUCE	NDTSTE/ NDUSTE	
Desktop/Server	Frameworks	78	78	78	78	78	78	78	78	56	56	68	68	**count**
		15.36	35.56	50.92	22.12%	77.88%	4.03	4.88	8.91	35.44%	64.56%	26.18%	12.09%	**mean**
		22.94	25.67	46.68	16.25%	16.25%	7.53	7.03	13.94	27.98%	27.98%	27.69%	13.17%	**std**
		0	2	3	0%	33.33%	0	0	0	0%	0%	0%	0%	**min**
		5	29	37	18.98%	81.02%	1	2	3	38.60%	61.40%	20.84%	7.90%	**50%**
		138	133	252	66.67%	100%	34	34	65	100%	100%	100%	53.12%	**max**
	Libraries	34	34	34	34	34	34	34	34	29	29	31	31	**count**
		14.65	41.56	56.21	17.91%	82.09%	4.18	6.35	10.53	26.88%	73.12%	34.16%	17.31%	**mean**
		27.58	52.94	78.92	13.08%	13.08%	9.68	7.16	15.79	22.24%	22.24%	31.58%	15.32%	**std**
		0	1	1	0%	42.48%	0	0	0	0%	22.22%	0%	0%	**min**
		4.50	30.50	34.50	17.03%	82.97%	1	4.50	5.50	27.27%	72.73%	33.33%	15.38%	**50%**
		143	317	460	57.52%	100%	56	30	86	77.78%	100%	100%	70%	**max**
	Tools	90	90	90	90	90	90	90	90	60	60	64	64	**count**
		8.82	29.97	38.79	12.95%	87.05%	2.37	2.40	4.77	32.67%	67.33%	28.75%	8.14%	**mean**
		25.64	22.92	38.89	14.82%	14.82%	11.25	3.58	12.53	36.29%	36.29%	36.93%	9.40%	**std**
		0	3	3	0%	11.28%	0	0	0	0%	0%	0%	0%	**min**
		2.50	26	27.50	8.86%	91.14%	0	1	1	26.79%	73.22%	11.80%	3.85%	**50%**
		228	135	257	88.72%	100%	104	17	109	100%	100%	100%	31.71%	**max**
	All	202	202	202	202	202	202	202	202	145	145	163	163	**count**
		12.33	34.08	46.41	17.33%	82.67%	3.31	4.02	7.34	32.58%	67.42%	28.71%	11.53%	**mean**
		25.05	31.03	50.77	15.62%	15.62%	9.69	5.95	13.80	30.73%	30.73%	32.26%	12.67%	**std**
		0	1	1	0%	11.28%	0	0	0	0%	0%	0%	0%	**min**
		4	27	34	14.82%	85.18%	0	2	2	30.43%	69.57%	20%	7.69%	**50%**
		228	317	460	88.72%	100%	104	34	109	100%	100%	100%	70%	**max**
Mobile	Frameworks	50	50	50	50	50	50	50	50	17	17	30	30	**count**
		8.40	23.16	31.56	12.01%	87.99%	1.22	2.98	4.20	22.38%	77.62%	10.33%	9.35%	**mean**
		28.57	30.19	55.89	14.33%	14.33%	3.82	7.25	10.61	28.90%	28.90%	20.57%	14.12%	**std**
		0	1	1	0%	45.40%	0	0	0	0%	0%	0%	0%	**min**
		1	11	12	6.70%	93.30%	0	0	0	10%	90%	0%	0%	**50%**
		184	153	337	54.60%	100%	22	36	58	100%	100%	66.67%	55.56%	**max**
	Libraries	60	60	60	60	60	60	60	60	24	24	24	24	**count**
		2.02	12.17	14.18	6.43%	93.57%	0.33	1.45	1.78	13.13%	86.87%	14.59%	12.20%	**mean**
		5.87	13.86	19.31	9.11%	9.11%	1.07	2.75	3.52	23.68%	23.68%	26.20%	14.89%	**std**
		0	1	1	0%	67.69%	0	0	0	0%	0%	0%	0%	**min**
		0	8	9	0%	100%	0	0	0	0%	100%	0%	9.09%	**50%**
		42	88	130	32.31%	100%	6	13	14	100%	100%	100%	53.33%	**max**
	Tools	42	42	42	42	42	42	42	42	7	7	12	12	**count**
		1.40	15.10	16.50	3.93%	96.07%	0.07	0.33	0.40	16.67%	83.33%	12.78%	3.52%	**mean**
		3.31	12.32	14.86	7.30%	7.30%	0.26	0.93	1.11	23.57%	23.57%	29.47%	9.01%	**std**
		0	1	1	0%	73.33%	0	0	0	0%	50%	0%	0%	**min**
		0	12	12	0%	100%	0	0	0	0%	100%	0%	0%	**50%**
		16	51	60	26.67%	100%	1	5	6	50%	100%	100%	31.25%	**max**
	All	152	152	152	152	152	152	152	152	48	48	66	66	**count**
		3.95	16.59	20.54	7.58%	92.42%	0.55	1.64	2.20	16.92%	83.08%	12.33%	9.33%	**mean**
		17.06	20.85	35.78	11.15%	11.15%	2.33	4.62	6.63	25.45%	25.45%	24.13%	13.80%	**std**
		0	1	1	0%	45.40%	0	0	0	0%	0%	0%	0%	**min**
		0	9	10	0%	100%	0	0	0	0%	100%	0%	0%	**50%**
		184	153	337	54.60%	100%	22	36	58	100%	100%	100%	55.56%	**max**
Multi-platform	Frameworks	22	22	22	22	22	22	22	22	18	18	18	18	**count**
		22.18	46	68.18	21.37%	78.63%	10.77	10.05	20.82	36.74%	63.26%	42.36%	21.46%	**mean**
		42.68	39.17	78.92	15.19%	15.19%	25.19	11.81	35.35	30.59%	30.59%	29.61%	16.37%	**std**
		0	4	4	0%	45.45%	0	0	0	0%	0%	0%	0%	**min**
		11	35	46	23.53%	76.47%	2	7.50	9.50	34.52%	65.48%	50%	25.90%	**50%**
		204	170	374	54.55%	100%	116	44	160	100%	100%	100%	57.14%	**max**
	Libraries	35	35	35	35	35	35	35	35	29	29	26	26	**count**
		6.86	26.57	33.43	12.23%	87.77%	2.23	6.54	8.77	25.39%	74.61%	52.26%	26.53%	**mean**
		12.88	17.92	28.30	13.16%	13.16%	3.39	6.80	9.36	24.95%	24.95%	34.22%	20.70%	**std**
		0	2	2	0%	50%	0	0	0	0%	0%	0%	0%	**min**
		2	24	26	10%	90%	1	5	7	18.18%	81.82%	50%	19.44%	**50%**
		61	78	122	50%	100%	14	29	41	100%	100%	100%	78.95%	**max**
	Tools	6	6	6	6	6	6	6	6	5	5	6	6	**count**
		8.33	30.67	39	19.88%	80.12%	1.67	1.67	3.33	50.77%	49.23%	26.39%	3.82%	**mean**
		10.41	28.01	37.68	11.35%	11.35%	2.73	2.42	4.84	50.03%	50.03%	29.07%	6.49%	**std**
		1	7	11	6.98%	63.64%	0	0	0	0%	0%	0%	0%	**min**
		3.50	23.50	27.50	20.84%	79.16%	0.50	0.50	1.50	53.85%	46.15%	25%	1.25%	**50%**
		28	80	108	36.36%	93.02%	7	6	13	100%	100%	58.33%	16.67%	**max**
	All	63	63	63	63	63	63	63	63	52	52	50	50	**count**
		12.35	33.75	46.10	16.15%	83.85%	5.16	7.30	12.46	31.76%	68.24%	45.59%	21.98%	**mean**
		27.74	29.02	54.14	14.25%	14.25%	15.46	8.89	22.66	30.28%	30.28%	32.57%	19.17%	**std**
		0	2	2	0%	45.45%	0	0	0	0%	0%	0%	0%	**min**
		3	28	36	13.33%	86.67%	1	4	7	22.50%	77.50%	50%	17.80%	**50%**
		204	170	374	54.55%	100%	116	44	160	100%	100%	100%	78.95%	**max**

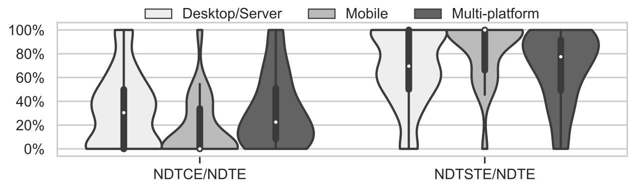


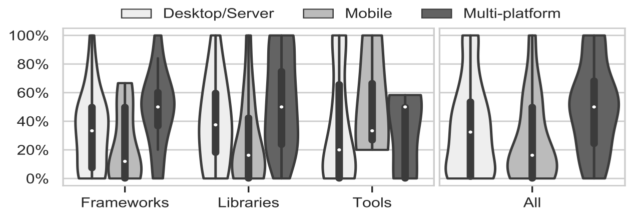
Figure 5: Ratios of NDTCE/NDTE and NDTSTE/NDTE.

NDUSTE/NDUE) and that there are more standard/third party exceptions being tested (i.e., ratios NDTCE/NDTE and NDTSTE/NDTE). Notice that we obtain the first two ratios by analyzing exclusively the SUT. Also, we obtain the other two ratios by analyzing exclusively the test methods. Now our intention is to understand, given the distinct used exceptions throughout the SUT, how many have associated test methods?

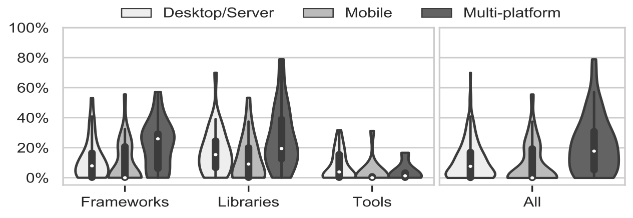
To do so, we take the Number of Distinct Tested Custom Exceptions (NDTCE) and divide by the Number of Distinct Used Custom Exceptions (NDUCE). Likewise, we divide the Number of Distinct Tested Standard/Third-party Exceptions (NDTSTE) by the Number of Distinct Used Standard/Third-party Exceptions (NDUSTE). To better explain these ratios, consider the ghidra project. This project has 138 distinct used custom exceptions (NDUCE) and 114 distinct used standard/third-party exceptions (NDUSTE). Regarding the tests, we have 33 tested custom exceptions (NDTCE) and 16 tested standard/third-party exceptions (NDTSTE). This way, when calculating the ratios we achieve the following: $33/138 = 23.91\%$ and $16/114 = 14.03\%$. Notice that, despite the relatively close numbers of distinct used custom exceptions (i.e., 138) and standard/third-party (i.e., 114), the test methods cover more custom exceptions than standard/third-party ones. Once again, we discarded projects that lead to a division by zero in columns NDTCE/NDUCE or NDTSTE/NDUSTE, and the final number of projects in each platform and domain is presented at the row "projects" in Table 3. Figure 6 illustrates the distribution of these ratios. According to our results, in 141 out of 254 (55.51%) projects the tests cover more custom exceptions than standard/third-party ones. However, the ratio in favor of custom exceptions is higher when considering the desktop/server (92 out of 149, i.e., 61.74%) and multi-platform projects (34 out of 52, i.e., 65.38%). The opposite happens for the mobile platform, where only 15 out of 53 (28.30%) projects cover more custom exceptions. Only six out of 254 (2.36%) projects (i.e., mockito, spring-batch, vavr, spring-data-redis, thumbnailator, and RxJava) have test methods that cover more than 50% of both the distinct used custom and standard/third-party exceptions.

Thus, our results suggest that, for the projects we analyzed, developers tend to create more test methods for distinct custom exceptions than for distinct standard/third-party exceptions in two platforms (i.e., desktop/server and multi-platform). We performed a statistical test to check if this difference is statistically significant.

We used the ratios NDTCE/NDUCE and NDTSTE/NDUSTE as input. First, we applied the Shapiro-Wilk test [23] to formally test for normality in each platform. After applying this test, we verified that our data do not follow a normal distribution. Therefore we applied the Mann-Whitney U Test [1]. We follow the convention of considering a factor as being significant to the response variable when p-value < 0.05. For the desktop/server and multi-platform projects we have significant differences (i.e., p-value 1.07^{-3} and 1.30^{-4}, respectively) between the cover ratios of custom and standard/third-party exceptions. However, the same result cannot be observed in the mobile platform, in which no significant statistical differences was found (i.e., p-value 0.25).



(a) Custom Exceptions (NDTCE/NDUCE).

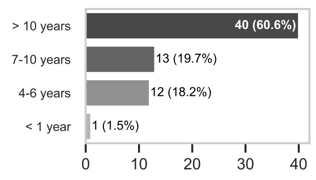


(b) Standard/Third-party Exceptions (NDTSTE/NDUSTE).

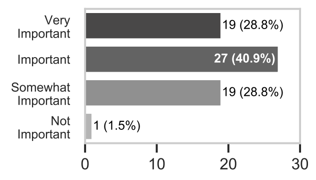
Figure 6: Ratios of NDTCE/NDUCE and NDTSTE/NDUSTE.

RQ3: How do developers perceive the exceptional behavior testing?

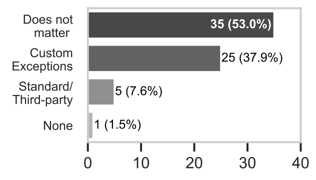
Overall, 66 developers completed our survey. Figure 7 summarizes the survey results. The majority of the respondents (69.70%) considers exceptional behavior testing as important. Moreover, 37.90% of the participants prioritize custom exceptions over standard/third-party, which is also in accordance to the findings of RQ2, in particular when considering desktop/server and multi-platform projects.



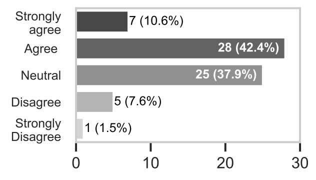
(a) Java experience.



(b) Exceptional tests importance.



(c) Prioritization.



(d) "Developers tend to neglect."

Figure 7: Survey Answers.

Regarding the sentence "*Software developers neglect tests that focus on exceptional behavior,*" the majority (53%) of the participants agrees with it. This way, the results are in sharp agreement with the findings of RQ1. Also, some developers left some comments on this sentence, and we noticed that some of these are similar. To better understand them, two researchers independently read the comments. Then, they agreed that 34 out of 39 comments fit into 8 categories. Comments with disagreements with respect to which category they belong to were discarded. Figure 8 presents the comments according to these categories in terms of a word cloud.

As can be observed, the majority of respondents stated that developers usually write tests for the "Happy Paths."

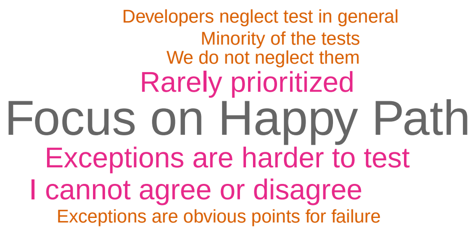


Figure 8: Word Cloud presenting 8 categories of the comments.

> "*It's much easier to write tests that check the success case. Writing tests for failure cases is a little harder.*"
> "*Developers usually focus on the common or 'good' cases.*"
> "*Most applications only do sunshine tests.*"

Also, we found comments that highlight the low importance given by some developers for the exceptional behavior testing.

> "*The least important of all tests.*"
> "*Exceptional behavior testing is rarely prioritized.*"
> "*It's often ignored because of the assumption that it only applies to an edge case.*"

On the other hand, we also found participants that mentioned that exceptional behavior testing cannot be neglected.

> "*My team at least does not neglect them.*"
> "*Exceptions are [...] some of the first things that I consider when writing unit tests.*"
> "*In my team we pay attention to it.*"

5 THREATS TO VALIDITY

The tool developed and applied in this study is not able to identify all the exceptions used in the `throw` instructions, `throws` clauses, and `catch` blocks, as well as in JUnit, TestNG, and AssertJ exception-testing constructs. This is due to the fact that such instructions allow the use of Superclasses (e.g., `Exception` or `RuntimeException`), and other object-oriented complex features such as polymorphism, inheritance, reflection, or generic types. In addition, we observed that both the source code and the test methods might be structured in different ways, which hinder their parsability (e.g., a `throw` instruction in which the exception is wrapped in a method call and a `throws` clause parameterized with a generic type). To sum up, our tool was not able to properly identify the exceptions used in approximately 3.50% of the exception-handling constructs, and in approximately 4.50% of test methods.

Also, our projects might have tests written using testing frameworks not covered by our tool. Nevertheless, we focus on very common Java testing technologies (i.e., JUnit, TestNG, and AssertJ). Thus, we do not expect major differences in the results.

Our selection process lies in the use of the GitHub query API and in the number of stars. The number of stars is a strong indicator on the number of developers interested in the project [24]. However, if the number of stars of some projects has been inflated by, for instance, the use of automated tools, projects with little relevance may have been included as objects of this study. We mitigate this threat by employing a criterion to assess whether the selected project has automated tests.

Our classification per platform and domain may represent a threat. Besides the use of a website that relies on maven dependencies and of a GitHub query, two researchers checked the classification independently, minimizing this threat.

Our survey relies on the "`test`" word to select potential participants. This way, we may select developers not experienced in tests, since the word is too general and may not be related to the scope of this paper. However, the participants of our survey reported they have great experience in Java and in Java testing frameworks.

6 RELATED WORK

Previous works aim to extend the coverage of testing for exception handling constructs. Goffi *et al.* [13] presented *Toradocu*, a tool that automatically generates tests from comments extracted from *Javadoc*. Also, they conducted a study based on 10 open source Java libraries and concluded that developers "*do not pay equal attention to testing exceptional behavior.*" To conclude that, they used one metric, i.e., they computed the `throw` statement coverage in comparison to other code instructions. They observed that the `throw` statement coverage is usually significantly low. Bernardo *et al.* [3] proposed an agile approach to define exceptional behavior of a system throughout the software development processes. They claim that "*manually-written test suites tend to neglect exceptional behavior.*" Despite these claims, the objective of both works [3, 13] is not to analyze whether and how developers test the exceptional behavior, i.e., they neither provided an in-depth investigation as we do nor a study to better understand the developer's thoughts. Differently, this is our main focus. So, we analyze 417 open source Java projects, use several metrics (collected based on parsing activities), and also compare custom and standard/third-party exceptions. Also, our study investigates whether there are differences in our numbers with respect to software platforms and domains. Finally, we also conduct a survey to confirm our quantitative results.

Romano *et al.* [20] used a genetic algorithm that evolves a population of test data to cover paths between input parameters and code statements that throws potential null pointer exceptions. Other works also try to extend the coverage of exception code, but with support of fault injection. Fu *et al.* [12] developed a compiler-directed fault injection static analysis to support white-box coverage testing of exception handlers. Martins *et al.* [15] presented *VerifyEx*, a tool that uses code instrumentation to exercise exception-handling constructs to increase the coverage rate when testing exceptional behavior. These works provide tools to automatically or semi-automatically improve the generation of tests for exception handling constructs. We also provide a tool, but to collect metrics regarding not only exception-handling constructs but also exception-testing constructs.

Osman *et al.* [18] performed an analysis on 90 Java projects and evaluated how developers use the different types of exceptions (custom, standard, and third-party). Differently, we checked if the tests cover more custom exceptions or standard/third-party ones.

7 CONCLUDING REMARKS

We performed an empirical study to analyze whether and how developers test the exceptional behavior in practice. To do so, we implemented a tool to collect several metrics related to exception-handling constructs; and exceptional-testing constructs. We performed our study in 417 open source Java projects from three platforms and three domains.

We found that 254 out of 417 (60.91%) projects have at least one test method dedicated to exceptional behavior. To better analyze this scenario, we also compute the ratio of the number of exceptional behavior test methods to the total number of test methods. We found that this ratio lies between 0% and 10% in 317 (76.02%) projects. Regarding used exceptions in the SUT, 239 (57.31%) projects test only up to 10% of them. We found that mobile developers in general pay less attention to exceptional behavior tests when compared to desktop/server and multi-platform developers. We also noticed that libraries have more exceptional behavior test methods when compared to frameworks and tools. We found more test methods covering custom exceptions over standard/third-party exceptions in desktop/server and multi-platform projects. Our statistical tests showed that this difference is significant in both platforms. Finally, we also conducted a survey to triangulate our results. In general, the collected answers confirm our findings.

We conclude that exceptional behavior testing is rare and indeed might be considered an exception. However, when considering multi-platform projects and libraries, the scenario is a bit better and these tests might not be so rare. Since developers tend to neglect exceptional behavior tests, one potential direction to improve this scenario is the use of automatic test suite generation tools.

As future work, we intend to execute the projects' test suites. So, we would not consider distinct exceptions, but all the locations they appear in the SUT. We also intend to consider other testing frameworks, and one more domain: applications.

REFERENCES

[1] Theodore W. Anderson and Jeremy D. Finn. 1996. *The new statistical analysis of data.* Springer.

[2] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. 2016. How Developers Use Exception Handling in Java?. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16).* ACM Press, 516–519.

[3] Rafael D. Bernardo, Ricardo Sales Jr., Fernando Castor, Roberta Coelho, Nelio Cacho, and Sergio Soares. 2011. Agile Testing of Exceptional Behavior. In *Proceedings of the 25th Brazilian Symposium on Software Engineering (SBES '11).* 204–213.

[4] Bruno Cabral and Paulo Marques. 2007. Exception Handling: A Field Study in Java and .NET. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP '07).* Springer, 151–175.

[5] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. 2018. How Swift developers handle errors. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '15).* ACM Press, 292–302.

[6] Byeong-Mo Chang and Kwanghoon Choi. 2016. A Review on Exception Analysis. *Information and Software Technology* 77, C (sep 2016), 1–16.

[7] Guilherme B. de Pádua and Weiyi Shang. 2017. Studying the Prevalence of Exception Handling Anti-Patterns. In *Proceeedings of the 25th International Conference on Program Comprehension (ICPC '17).* 328–331.

[8] Guilherme B. de Pádua and Weiyi Shang. 2018. Studying the Relationship between Exception Handling Practices and Post-Release Defects. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18).* ACM Press, 564–575.

[9] Keith Ó Dúlaigh, James F. Power, and Peter J. Clarke. 2012. Measurement of Exception-Handling Code: An Exploratory Study. In *Proceedings of the 5th International Workshop on Exception Handling (WEH '12).* IEEE Press, 55–61.

[10] Engineering and Systems Software Research Group (EASY). 2020. Research Replication Package. https://github.com/easy-software-ufal/exceptionhunter

[11] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale Analysis of Framework-specific Exceptions in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18).* ACM Press, 408–419.

[12] Chen Fu, Ana L. Milanova, Barbara G, Ryder, and David G. Wonnacott. 2005. Robustness testing of Java server applications. *IEEE Transactions on Software Engineering* 31, 4 (2005), 292–311.

[13] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16).* 213–224.

[14] Cheng-Ying Mao and Yan-Sheng Lu. 2005. Improving the robustness and reliability of object-oriented programs through exception analysis and testing. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '05).* IEEE Press, 432–439.

[15] Alexandre L. Martins, Simone Hanazumi, and Ana C.V. de Melo. 2014. Testing Java Exceptions: An Instrumentation Technique. In *IEEE 38th International Computer Software and Applications Conference Workshops (COMPSACW '14).* 626–631.

[16] Hugo Melo, Roberta Coelho, and Christoph Treude. 2019. Unveiling Exception Handling Guidelines Adopted by Java Developers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19).* 128–139.

[17] Taiza Montenegro, Hugo Melo, Roberta Coelho, and Eiji Barbosa. 2018. Improving developers awareness of the exception handling policy. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18).* 413–422.

[18] Haidar Osman, Andrei Chis, Claudio Corrodi, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Exception Evolution in Long-lived Java Systems. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17).* IEEE Press, 302–311.

[19] Darrell Reimer and Harini Srinivasan. 2003. Analyzing exception usage in large Java applications. In *Workshop on Exception Handling in Object Oriented Systems (EHOOS '03).*

[20] Daniele Romano, Massimiliano Di Penta, and Giuliano Antoniol. 2011. An Approach for Search Based Testing of Null Pointer Exceptions. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST '11).* IEEE Press, 160–169.

[21] Barbara G. Ryder, Donald Smith, Ulrich J. Kremer, Michael D. Gordon, and Nirav Shah. 2000. A Static Study of Java Exceptions Using JESP. In *Proceedings of the 9th International Conference on Compiler Construction (CC '00).* Springer, 67–81.

[22] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study. In *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR '16).* ACM Press, 212–222.

[23] Samuel S. Shapiro and B. Wilk Martin. 1965. An Analysis of Variance Test for Normality. *Biometrika* 52 (1965), 591–611.

[24] Hudson Silva and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129.

[25] Saurabh Sinha and Mary Jean Harrold. 2000. Analysis and Testing of Programs with Exception Handling Constructs. *IEEE Transactions on Software Engineering* 26, 9 (2000), 849–871.

[26] Rebecca J. Wirfs-Brock. 2006. Toward Exception-Handling Best Practices and Patterns. *IEEE Press* 23, 5 (2006), 11–13.

[27] Pingyu Zhang and Sebastian Elbaum. 2014. Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain. *Transactions on Software Engineering and Methodology* 23, 4, Article 32 (2014), 32:1–32:28 pages.