UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS GRADUAÇÃO EM INFORMÁTICA

FRANCISCO DALTON BARBOSA DIAS

É uma Exceção Testar Comportamento Excepcional? Uma Avaliação Empírica Utilizando Testes Automatizados em Java

Maceió-AL

Maio de 2020

FRANCISCO DALTON BARBOSA DIAS

É uma Exceção Testar Comportamento Excepcional? Uma Avaliação Empírica Utilizando Testes Automatizados em Java

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Supervisor: Márcio de Medeiros Ribeiro

Maceió-AL

Maio de 2020

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

D541e Dias, Francisco Dalton Barbosa.
É uma exceção testar comportamento excepcional? : uma avaliação empírica utilizando testes automatizados em java / Francisco Dalton Barbosa Dias. – 2020.
53 f. : il.

Orientador: Márcio de Medeiros Ribeiro.
Dissertação (mestrado em Informática) - Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2020.

Texto em inglês.
Bibliografia: f. 43-46.
Apêndice: f. 48-53.

1. Java (Linguagem de programação de computador) - Comportamento excepcional. 2. *Software* - Testes. I. Título.

CDU: 004.43

AGRADECIMENTOS

Assim como já fiz no passado, começo agradeço primeiramente a Deus por continuar a me dar forças para continuar andando, mesmo quando cançado, para alcançar meus objetivos e, principalmente, por permitir que eu os alcance.

Mais uma vez agradeço a Isadora Tenório, que durante meu TCC era minha namorada e hoje é minha esposa, pela compreensão pelos inúmeros dias em que eu fui completamente ausente enquanto eu estudava ou trabalhava em minha pesquisa, por suportar ter de ir dormir tarde e acordar cedo por causa dos meus horários malucos, por toda a força, apoio, carinho e amor que vem me dando há tantos anos!

Agradeço (e muito!!) ao professor e meu orientador, Márcio Ribeiro, por ter aceitado a me ter novamente como orientando e continuado a me ajudar, mesmo quando eu não estava dando motivos para isso. Por ter sido de extrema importância durante a minha passagem pela graduação e mais ainda durante a do mestrado.

Agradeço aos membros da banca que foram de extrema importância na evolução do meu trabalho de pesquisa e que se dispuseram a participar deste momento tão importante na minha vida, mesmo no meio de uma pandemia.

Após quase quatro anos desde que me tornei bacharel em Ciência da Computação, posso repetir: agradeço aos professores do IC que, acima de tudo, mostraram que são educadores preocupados com seus alunos e que passaram conhecimento e sabedoria muito além do habitual da sala de aula.

Por fim, agradeço a todos os que fazem do Instituto de Computação da UFAL o que ele é!

Obrigado!

RESUMO

Executar testes de *software* é uma atividade crucial para avaliar a qualidade interna de um sistema. Durante os testes, os desenvolveres geralmente criam testes para o comportamento esperado de uma determinada funcionalidade (*e.g.,* o arquivo foi corretamente enviado para a nuvem?). No entanto, pouco é conhecido se os desenvolvedores também criam testes para comportamentos excepcionais (*e.g.*, o que acontece se a conexão de rede for interrompida enquanto o arquivo é enviado?). Para minimizar essa lacuna de conhecimento, neste trabalho nós desenhamos e executamos um estudo de método misto para entender se e até que ponto 417 projetos Java de código aberto estão testando o comportamento excepcional usando os *frameworks* JUnit e TestNG, e a biblioteca AssertJ. Através de uma análise estática, nós descobrimos que 254 (60,91%) projetos possuem ao menos um método de teste dedicado ao comportamento excepcional. Também descobrimos que o número de métodos de testes para o comportamento excepcional em relação ao total de testes está entre 0% e 10% em 317 (76,02%) projetos. Além disso, 239 (57,31%) projetos testam apenas até 10% das exceções usadas no código do sistema em teste—*System Under Test (SUT)*—. Quando avaliamos aplicativos para dispositivos móveis, nós observamos que, em geral, os desenvolveres dedicam menos atenção aos testes de comportamentos excepcionais quando comparados aos desenvolvedores de aplicações para *desktop*/servidores e multi-plataforma. Em geral, nós encontramos mais métodos de testes cobrindo exceções customizadas (as que são criadas dentro do próprio projeto) do que as exceções padrões disponíveis no *Java Development Kit (JDK)* ou em bibliotecas de terceiros. Além disso, nós também realizamos uma análise dinâmica em um subconjunto de 39 projetos, com dados de cobertura de linha publicamente disponíveis, para investigar se e até que ponto as suítes de testes excercitam os `throw` *statements* encontrados no código do sistema testado. Os resultados da nossa análise dinâmica indicam que as suítes de testes não exercitam a maioria dos `throw` *statements*, mesmo em projetos onde parece haver preocupações relacionadas à cobertura de código. Nós também enriquecemos o entendimento sobre como os desenvolvedores escrevem seus testes de comportamentos excepcionais em termos de construções do JUnit, TestNG, e do AssertJ. Para triangular os resultados, nós conduzimos uma pesquisa com 66 desenvolvedores dos projetos que estudamos. Em geral, os resultados da pesquisa confirmaram nossas descobertas. Em particular, a maioria dos participantes concordam que os desenvolvedores frequentemente negligenciam testes de comportamentos excepcionais. Como implicações, nossos números podem ser importantes para alertar os desenvolvedores de que mais esforço deve ser feito na criação de testes para comportamentos excepcionais.

Palavras-chaves: Exceções, Comportamento Excepcional, Testes de Software.

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 work we designed and performed a mixed-method study to understand whether and to what extent 417 open source Java projects are testing the exceptional behavior using the JUnit and TestNG frameworks, and the AssertJ library. Through static analysis, 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. Moreover, we also performed a dynamic analysis on a subset of 39 projects, with publicly available line coverage reports, to investigate whether and to what extent the test suites exercises the `throw` statements found in the SUT. The results of the dynamic analysis indicate that the test suites do not exercise the most of `throw` statements, even in projects where there seem to have concerns about code coverage. We also enriched the understanding of how developers write their exceptional behavior tests in terms of constructs from JUnit, TestNG, and AssertJ. 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.

Keywords: Exceptions, Exceptional Behavior, Software Testing.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

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, 37, 6]. Hence, there are several studies that try to assess the quality of exception handling code [6, 4], development patterns [28, 33, 26, 9], or best practices and usage scenarios [5, 10, 29, 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 exceptional behavior testing [16, 3]. 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 [23, 10, 9]. In fact, studies provide evidences that the majority of crashes in Android Apps are related to exceptions defined in the Android Framework [14]. Thus, it is possible that a software system presents failures that could be otherwise avoided through more rigorous testing that handle exceptional behavior [19, 3], an activity that we call throughout this work 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, 10, 28, 33, 29, 22], 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 work we present a *mixed-method* study consisting of: (1) an empirical investigation based on the static analysis of 417 projects and dynamic analysis of 39 projects to understand whether and to what extent developers actually test the exceptional behavior; and (2) a survey with 66 developers from these projects to triangulate our quantitative results. 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

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

of October 2019). For each project, we selected and downloaded the latest version available. We categorized 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 [11] (*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 collected 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.

The results of our static analysis [8] indicates 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 68.12% of the projects than standard/third party exceptions. This is particularly the case of desktop/server (70.23%) and multi-platform (73.91%) projects.

Furthermore, we carried out a dynamic analysis on a subset of 39 projects, from our 417 studied projects, with publicly available line coverage reports. To do so, our tool automatically retrieved line coverage data publicly available on Coveralls[4] and Codecov[5] web services. Unfortunately, the static analysis limited our results to understand only how many of the used exceptions are tested (*e.g.,* Does the test suite exercise the `RuntimeException`?), and did not allowed us to identify whether and to what extent the `throw` statements spread throughout the SUT are actually exercised by the tests (*e.g.,* Does the test suite exercise every `throw new RuntimeException` statement?). The results of our dynamic analysis showed that 29 out of 39 (74.36%) projects report Line Coverage ratios greater than or equal to 60%. Nevertheless, only six (15.38%) projects achieve a coverage of `throw` statement lines greater than or equal to 60%. Our dynamic analysis also provided additional evidence that the custom exceptions have better coverage than the standard/third-party exceptions, reinforcing our static analysis results [8]. In addition, we noticed that a larger percentage of multi-platform projects and libraries have higher numbers to Throw Statement Line Coverage ratios. These results are also in accordance with our static analysis results [8]. Last but not least, in this work we also enrich the understanding of how developers write their exceptional behavior tests in terms of constructs from JUnit, TestNG, and AssertJ. We provide statistics regarding the usage of the constructs. We notice that the newer constructs designed to test the exceptional behavior created over the last years should be better

[4] `https://coveralls.io/`
[5] `https://codecov.io/`

spread throughout the developers community so that developers can make the task of testing exceptions less difficult and more efficient.

In this work, we have the following scope: we focus on Java automated tests written using JUnit, TestNG, or AssertJ; our tool collects metrics statically, and we retrieve publicly available line coverage data from online tools to perform our dynamic analysis; we collect all the exceptions used in the SUT (checked and unchecked). All projects we use in this study are open source.

2 BACKGROUND

2.1 Java Exceptional Behavior

Exception handling techniques are important in modern object-oriented software development. Actually, it is present in a large number of programming languages [18, 6]. 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, 37, 6].

In Java, the whole exception handling mechanisms are designed to attend a class hierarchy capable of representing different abnormal behaviors. Thus, all exceptions have the `Throwable` superclass as an ancestor. This hierarchy allows the creation of three kinds of exceptions [18]:

- **Unchecked Exceptions:** These exceptions have the superclass `RuntimeException` as an ancestor. As this type only occurs at runtime, they can not be checked at compilation time. Moreover, the developers are not obligated to handle this kind of exception;
- **Errors:** Are used by the Java Virtual Machine (JVM) to represent serious errors that can not be recovered. The class `Error` and all its subclasses are also classified as unchecked exceptions;
- **Checked Exceptions:** These exceptions have neither `RuntimeException` nor `Error` class as an ancestor. This kind of exception is checked at compile time and the developers must handle them.

It is also possible to classify exceptions by the place where they are defined [25].:

- **Custom Exceptions** The ones created in the own project;
- **Standard Exceptions:** Any exception defined in the Java Development Kit (JDK);
- **Third-party Exceptions:** Exceptions that belongs to neither the standard exceptions nor the custom exceptions.

2.2 Java Automated Testing Frameworks and Libraries

Testing frameworks and libraries are designed mainly to simplify the testing tasks. They usually provide mechanisms to quickly create and reproduce from unit tests to integration tests, among other features. In this work, we investigate tests written in two frameworks (*i.e.*, JUnit and TestNG), and one library (*i.e.*, AssertJ).

- **JUnit:**[1] It is the most well-known Java automated unit testing framework. With JUnit, it is possible to perform from unit tests to integration tests, among other types of tests. JUnit makes extensive use of annotations to facilitate the creation and specification of complex tests. Also, it provides features such as:
  - JUnit is an open source framework, which is used for writing and running tests.
  - Provides annotations to identify test methods.
  - Provides assertions for testing expected results.
  - Provides test runners for running tests.
  - JUnit tests allow you to write codes faster, which increases quality.
  - JUnit is elegantly simple. It is less complex and takes less time.
  - JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
  - JUnit tests can be organized into test suites containing test cases and even other test suites.
  - JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.
- **TestNG:**[2] It is designed to cover all categories of tests: unit, functional, end-to-end, integration, etc. Also, it is similar to the JUnit framework, but it claims to have some functionalities that make it more powerful and easier to use, such as:
  - Annotations.
  - Run your tests in arbitrarily big thread pools with various policies available (all methods in their own thread, one thread per test class,

    etc...).
  - Test that your code is multithread safe.
  - Flexible test configuration.
  - Support for data-driven testing (with @DataProvider).
  - Support for parameters.
  - Powerful execution model (no more TestSuite).
  - Supported by a variety of tools and plug-ins (Eclipse, IDEA, Maven, etc...).
  - Embeds BeanShell for further flexibility.

[1] `https://www.tutorialspoint.com/junit/junit_overview.htm`
[2] `https://testng.org/doc/`

  - Default JDK functions for runtime and logging (no dependencies).
  - Dependent methods for application server testing.

- **AssertJ:**[3] It can be used with both JUnit and TestNG. The focus of this library is not to provide a testing infrastructure, but to provide a rich set of assertions, truly helpful error messages, and improvements to test code readability. This library provides a rich set of assertions to test the exceptional behavior and for some of the most used libraries, such as:
  - A core module to provide assertions for JDK types (String, Iterable, Stream, Path, File, Map...)
  - A Guava module to provide assertions for Guava types (Multimap, Optional...)
  - A Joda Time module to provide assertions for Joda Time types (DateTime, LocalDateTime)
  - A Neo4J module to provide assertions for Neo4J types (Path, Node, Relationship...)
  - A DB module to provide assertions for relational database types (Table, Row, Column...)
  - A Swing module provides a simple and intuitive API for functional testing of Swing user interfaces

2.2.1 Exception-handling Constructs

The exception-handling constructs are responsible for changing the normal control flow to an alternative flow when an exceptional event occurs. Then, they are also responsible for returning the system to the normal flow after the event has been treated.

In Java, an exception is raised by the `throw` statement and handled by a `catch` block. If it is not caught by any `catch` block, then the `throws` clause may propagate the exception along the call chain until it is caught by a `catch` block or until it reaches the main method, where the execution is abnormally terminated.

2.2.2 Exception-testing Constructs

Since the exceptional behavior has some differences in comparison to the normal behavior, the testing frameworks and libraries have specific constructs designed to test the exceptional flow properly. For example, the JUnit have the `expected` attribute of the `@Test` annotation. With this attribute, the developer specifies that an exception should be thrown somewhere in the annotated test method.

[3] `https://assertj.github.io/doc/`

2.2.3 Code Coverage

Code Coverage is a popular software engineering technique that allows developers to track the quality of the test suite. This technique helps to determine how much of the System Under Test (SUT) the test suite exercises [24, 17, 39, 20, 12]. For example, it is possible to measure coverage by analyzing the number of exercised lines of code, methods, classes, statements, branches, and so forth [39, 24, 12, 39, 20]. There is no consensus as to which metric is the best one to use, but it is accepted that each one can provide different information with different utilities. Also, there is no consensus about the ideal coverage ratio value [20, 24]. Nevertheless, the maintainers of the EMMA tool[4]—a free Java code coverage tool— suggest in their FAQ that the *"Practice shows that coverage percentages below, say, 60-70% correspond to poorly tested software"* [12]. Even though, there are suggestions that a piece of software should only be released when it reaches at least 80% [12, 17] or 85% [36, 20] of code coverage. Some web services provide ways to easily view the coverage of each of the classes in a project. These web services present the entire file structure and the number of executions of each line. In addition, they allow the developer to configure the use of different coverage metrics. Two of the most important code coverage web services are: Coveralls[5] and Codecov.[6]

[4] `http://emma.sourceforge.net`
[5] `https://coveralls.io/`
[6] `https://codecov.io/`

3 MOTIVATING SCENARIO

We refer to "exceptional behavior testing" as test methods that expect exceptions to be raised. We illustrate an example in Listing 3.1. 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 3.1 – Exceptional behavior test method example.

Many studies on exceptions and error handling have been developed along the years [38, 3, 16]. 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.* [16] 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 3.1 was extracted from the eclipse-collections project.[1] To evaluate the exceptional behavior, this particular test method relies on the `expected` attribute of the `@Test` annotation (Line 1) 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[2] (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. Moreover, a single exception can be thrown from multiple lines in the code, which makes us wonder how many of these lines are actually executed by the test suites.

Given this scenario, although existing works [3, 16] claim that developers neglect exceptional behavior testing, they did not provide an in-depth investigation on whether and to what extent developers test the exceptional behavior. In this work, we perform both static and dynamic analysis of the projects, we consider a much higher number of projects (417 *versus* 10 [16]). We also identify differences among platforms and domains, correlate the results with repositories' characteristics, and triangulate our results with a survey with 66 participants.

[1] https://github.com/eclipse/eclipse-collections
[2] https://github.com/NationalSecurityAgency/ghidra

4 EMPIRICAL STUDY

In this chapter we present our empirical study. First, we introduce the research questions (Section 4.1). Then we present the studied projects and the criteria used to select them (Section 4.2). Afterwards, we detail the metrics we use to answer our research questions (Section 4.3). Also, we describe our tool used to collect and analyze data (Section 4.4). Finally, we explain the procedures we use to perform our survey (Section 4.5).

4.1 Goal and Research Questions

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

We intend to answer the following research questions:

- **RQ1:** To what extent do developers test the exceptional behavior with automated tests?
- **RQ2:** Do the test suites test more distinct custom exceptions or distinct standard/third-party exceptions?
- **RQ3:** To what extent the test suites cover `throw` statement lines?
- **RQ4:** Do the test suites cover more `throw` statement lines of custom exceptions or standard/third-party exceptions?
- **RQ5:** How do developers test the exceptional behavior in terms of exception-testing constructs?
- **RQ6:** 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 complement **RQ1** and helps us to better understand if developers are concerned about testing the multiple `throw` statements spread throughout the SUT. Answering **RQ4** complements **RQ2** in terms of a dynamic analysis. Answering **RQ5** and **RQ6** is important to comprehend how developers perceive the exceptional behavior testing and to raise developers' practices, thoughts, and opinions. This might help researchers and practitioners with developing processes and tools to focus on exceptional behavior tests.

4.2 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:

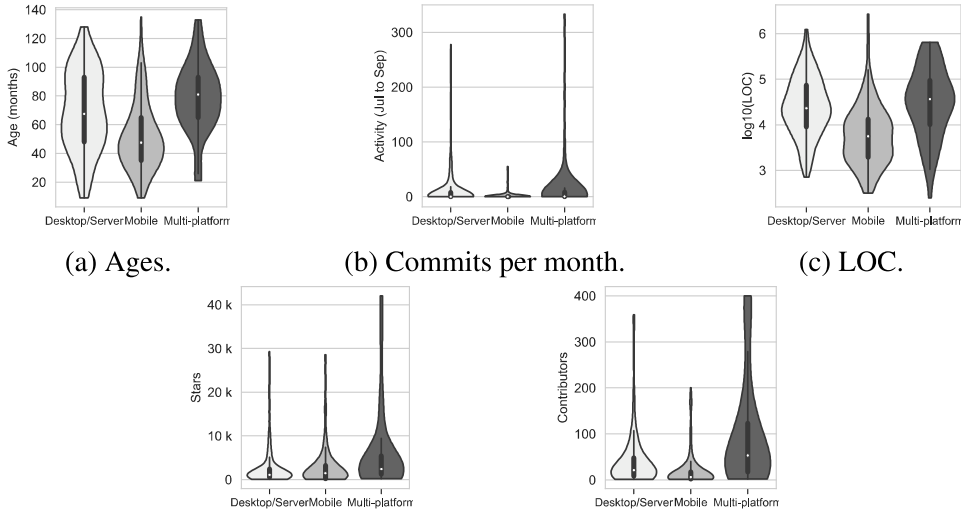
1. Has at least one custom, standard, or third-party exception being used in exception-handling constructs (*i.e.*, `throws` clauses, `throw` statements, or on `catch` blocks) in the SUT;
2. Has at least one test method using JUnit, TestNG, or AssertJ.

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 article 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[1] 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.

[1] `https://github.com/ReactiveX/RxJava`



(a) Ages. (b) Commits per month. (c) LOC.

(d) Stars. (e) Contributors.

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

4.3 Collected Metrics

1. *Number of Distinct Used Exceptions* ***(NDUE)***: Total number of distinct exceptions found in `throw` statements, `throws` clauses, and `catch` blocks in the SUT;

2. *Number of Distinct Used Custom Exceptions* ***(NDUCE)***: Total number of distinct custom exceptions found in `throw` statements, `throws` clauses, and `catch` blocks in the SUT;

3. *Number of Distinct Used Standard/Third-party Exceptions* ***(NDUSTE)***: Total number of distinct standard/third-party exceptions found in `throw` statements, `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;

9. *Number of Throw Statement Lines* ***(NTS)***: Total number of `throw` statements lines found in the SUT;

10. *Number of Throw Statement Lines of Custom Exceptions (**NTSCE**)*: Total number of `throw` statement lines of custom exceptions found in the SUT;

11. *Number of Throw Statement Lines of Standard/Third-party Exceptions (**NTSSTPE**)*: Total number of `throw` statements lines of standard/third-party exceptions found in the SUT;

12. *Number of Covered Throw Statement Lines (**NCTS**)*: Total number of `throw` statement lines exercised at least once by the test suite;

13. *Number of Covered Throw Statement Lines of Custom Exceptions (**NCTSCE**)*: Total number`throw` statement lines of custom exceptions exercised at least once by the test suite;

14. *Number of Covered Throw Statement Lines of Standard/Third-party Exceptions (**NCTSSTPE**)*: Total number `throw` statement lines of standard/third-party exceptions exercised at least once by the test suite;

15. *Line Coverage*: The ratio of lines of code covered by the test suite (retrieved from Coveralls or Codecov web services);

16. *Throw Statement Line Coverage*: The ratio of the Number of Covered Throw Statement Lines (**NCTS**) to the Number of Throw Statement Lines (**NTS**).

4.4 Tool Description and Usage Scenarios

We developed a tool [13] that analyzes Git repositories. For each repository, the tool performs a static analysis on Java code, and tries to retrieve projects' dynamic data from external web services (Coveralls and Codecov). This tool was based on JavaParser,[2] a lightweight library for supporting syntactic analysis in Java code, version 3.13.2.

4.4.1 Static Analysis

Our tool identifies exception-handling constructs [11] (*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` 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.

[2] `https://javaparser.org`

```
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 4.1 – Examples of exception-handling constructs.

To better explain how our tool works, consider the code snippet from the Spring-Framework[3] project shown in Listing 4.1. If this code was analyzed by our tool, the exception `TypeMismatchException` described in the `throws` clause (Line 1) would be collected. Moreover, the exceptions `ConvertedNotFoundException` (Line 4), `IllegalStateException` (Line 4), `ConversionException` (Line 6), and `IllegalArgumentException` (Line 6) would also be collected, because they are used inside `catch` blocks. Finally, the exceptions `ConversionNotSupportedException` (Line 5) and `TypeMismatchException` (Line 7) would also be collected because they are used within a `throw` instruction.

Listing 4.2 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 3), which in this case is the `IllegalStateException` exception.

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

Listing 4.2 – `assertThrows` method call example.

Similarly, we can collect the exceptions used in the `expected` attribute of the `@Test` annotations (see Listing 3.1, Line 1). Our tool can also collect the exceptions used with the `ExpectedException` rule. Listing 4.3 illustrates an example. In this case, the tool would collect the `IllegalArgumentException` exception (Line 5).

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

Listing 4.3 – `ExpectedException` rule example.

[3] 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

Our tool also collects the exceptions being used in the `catch` blocks right after a call to the `fail` method (Listing 4.4). To test the exceptional behavior, the developer expects the `TypeMismatchException` (Line 5) 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 4.4 – `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 4.1–4.4. 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.

4.4.2 Dynamic Analysis

Additionally, our tool also checks if the project under analysis has line coverage data publicly available on Coveralls or Codecov web services. When this is true, our tool extracts the project's line coverage ratio, and the coverage data for each line with a `throw` statement found in the SUT.

Figure 2 illustrates the coverage extraction process. We follow the steps described below.

1. Given the source code of a project, our tool identifies the classes and lines where `throw` statements are found;

2. In case the project has data available in Coveralls or Codecov, the tool retrieves the line coverage ratio of the whole project and individual coverage reports for each class. This

data is specific to the project version under analysis;

3. Now, each class is evaluated. In case the class has coverage data, the tool creates a report containing the number of times that the test suite exercised each line of the class.

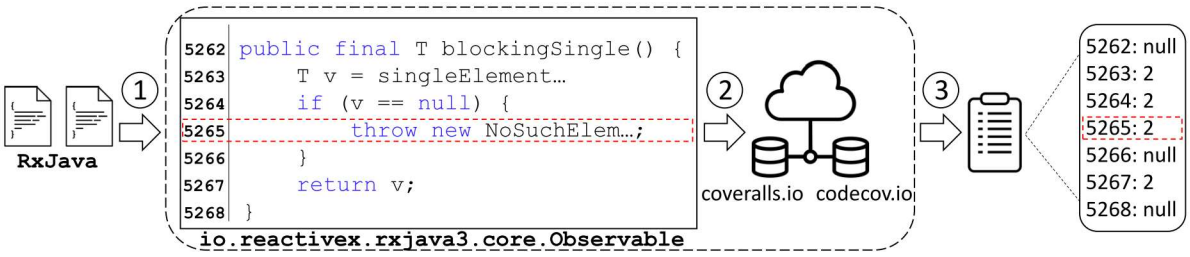


Figure 2 – Coverage extraction process.

In our illustrating example, the resulting report shows that a `null` value has been assigned to line 5262. This happens when a line of code is not relevant to the coverage analysis. On the other hand, the value two has been assigned to line 5265, where there is a `throw` statement. Because this number is greater than zero, the test suite exercised such a line.

4.5 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.

5 RESULTS AND DISCUSSION

In this chapter, we answer our research questions and discuss the results we obtained. Firstly, we present the results of our static analysis (Section 5.1). Then, we discuss the results of our dynamic analysis (Section 5.2). Also, we present the most widely used exception-testing constructs (Section 5.3). Finally, we present and discuss the implications of our survey's answers (Section 5.4).

All data, scripts and the tool created in this study are also online available in our companion website [13].

5.1 Static Analysis

In this section, we analyze exception-related data statically extracted by our tool from 417 projects. We use this data to answer our research questions **RQ1** and **RQ2**.

RQ1 To what extent do developers test the exceptional behavior with automated tests?

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 3 (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.

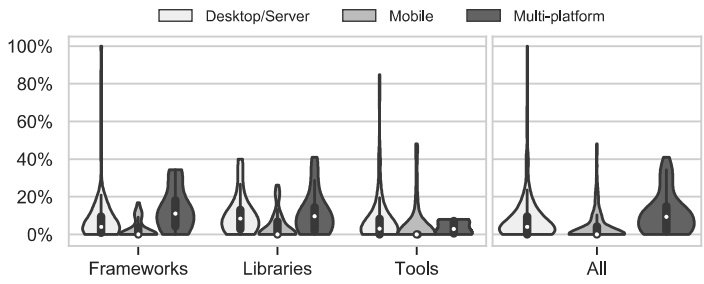


Figure 3 – Ratio of NEBTM/NTM.

Furthermore, for each project we also calculate the ratio of the Number of Distinct Tested Exceptions (NDTE) 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 4 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 NDTE/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.

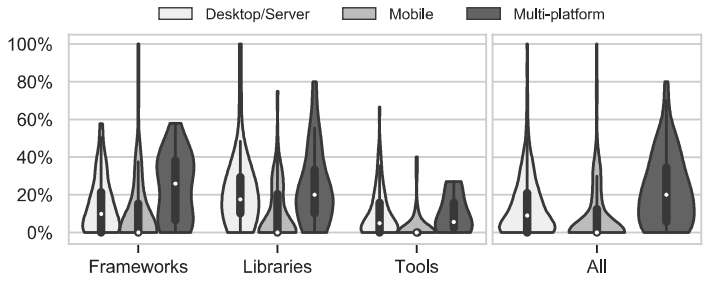


Figure 4 – Ratio of NDTE/NDUE.

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 [14]. 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 article.

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 suites test 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 5 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 "count" in Table 3.

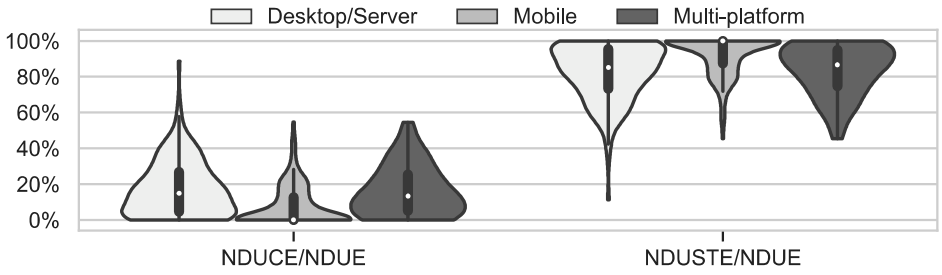


Figure 5 – Ratios of NDUCE/NDUE and NDUSTE/NDUE.

Table 4 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 "count" in Table 3 and in Table 4 have different numbers (*e.g.*,., in the first row (desktop/server frameworks), the number of projects dropped from 78 to 56 projects). Figure 6 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.

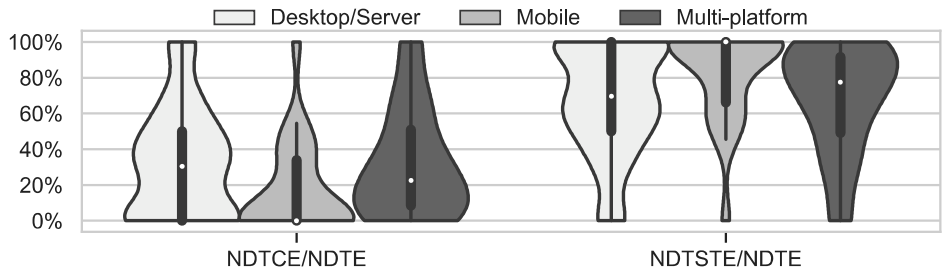
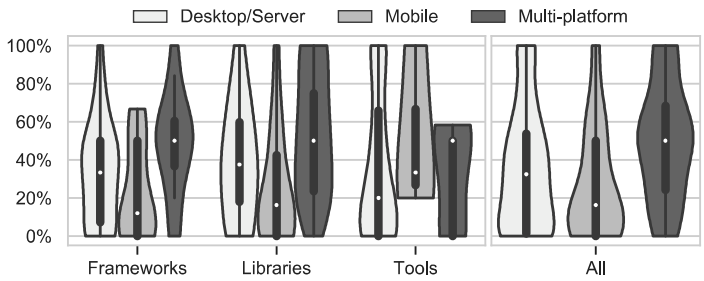


Figure 6 – Ratios of NDTCE/NDTE and NDTSTE/NDTE.

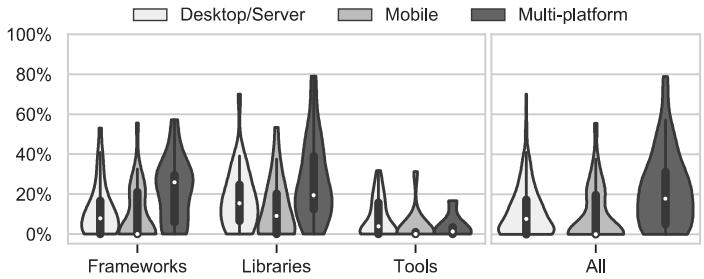
Our results suggest that standard/third party exceptions are more used throughout the SUT (*i.e.*, ratios NDUCE/NDUE and NDUSTE/NDUE) and that there are more standard/third party exceptions being tested (*i.e.*, ratios NDTCE/NDTE and NDTSTE/NDTE). Notice that we obtain the first two ratios by analyzing 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 "count" in Table 5. Figure 7 illustrates the distribution of these ratios. According to our results, in 141 out of 207 (68.12%) 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

131, *i.e.*, 70.23%) and multi-platform projects (34 out of 46, *i.e.*, 73.91%). A tie happens for the mobile platform, where 15 out of 30 (50%) projects cover more custom exceptions. Only six out of 207 (2.90%) 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.



(a) Custom Exceptions (NDTCE/NDUCE).



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

Figure 7 – Ratios of NDTCE/NDUCE and NDTSTE/NDUSTE.

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 [31] 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).

5.2 Dynamic Analysis

In this section, we use the code coverage data from a subset of projects to answer the research questions **RQ3** and **RQ4**. Only 48 projects of our initial dataset have line coverage publicly available in Coveralls or Codecov web services. However, we discarded nine projects due to incorrect configuration of these tools. Thus, to answer **RQ3** and **RQ4**, we restrict our analysis to only 39 out of 417 (9.35%) projects. The new distribution of this dataset is: 26 (66.67%) desktop/server projects (11 frameworks, three libraries, and 12 tools); five (12.82%) mobile projects (one framework, three libraries, and one tool); and eight (20.51%) multi-platform projects (one framework, five libraries, and two tools). As can be observed in Figure 8, the majority of the projects—29 out of 39 (74.36%)—report Line Coverage ratios greater than or equal to 60%. We use this number as a reference because coverage tools [12] report that percentages below 60-70% correspond to poorly tested software. It is important to emphasize that we retrieve these ratios from Coveralls and Codecov web services.

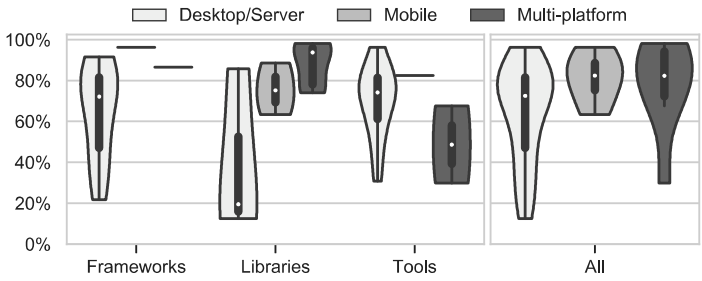


Figure 8 – Line coverage ratios.

RQ3: To what extent the test suites cover `throw` statement lines?

In our dataset, there are projects like RxJava, shown in Figure 9, that exercises almost every `throw` statement line, regardless of the thrown exception. This might be an indicator that the RxJava developers pay attention to writing exceptional behavior tests. However, most projects have results like the ones found in the Caffeine[1] project (see Figure 10), where the majority of the `throw` statements are not exercised by the test suite. For example, the `UnsupportedOperationException` is identified in 65 `throw` statements, but only four of these statements are exercised by the test suite. This scenario happens with many other exceptions of the project.

To better understand whether and to what extent the test suites exercises each `throw` statement lines, we calculate the ratio of the Number of Covered Throw Statement Lines (NCTS) to the Number of Throw Statement Lines (NTS). We name this ratio as Throw Statement Line Coverage (see Table 6). Figure 11 illustrates the distribution of these ratios for all projects. The median values of each platform are 15.67% for desktop/server, 37.50% for mobile, and 30.16%

[1] `https://github.com/ben-manes/caffeine`

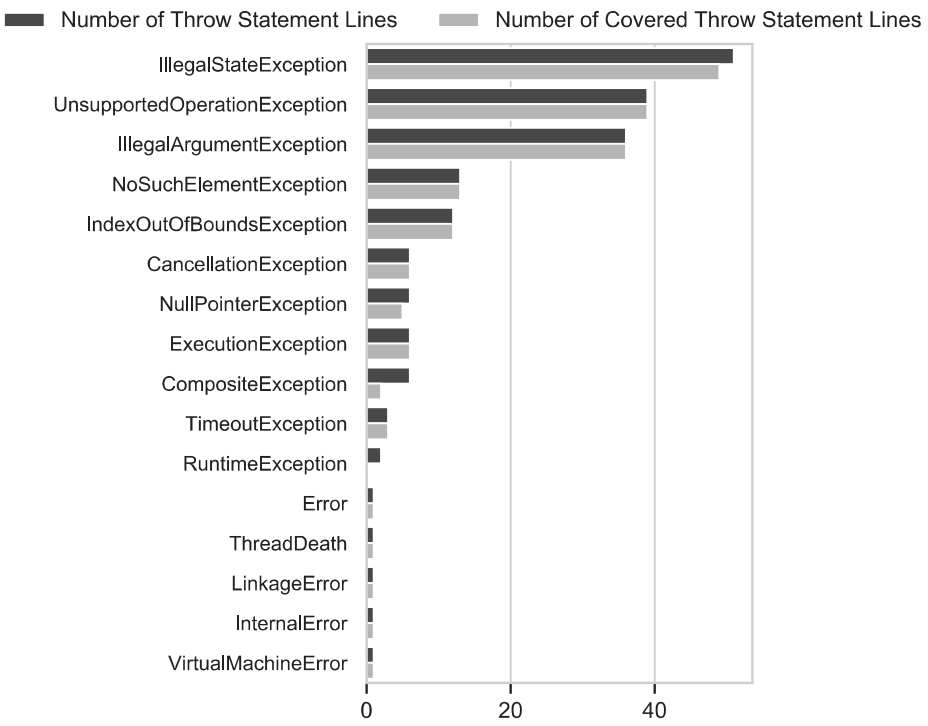


Figure 9 – RxJava's throw statements data.

for multi-platform. However, these numbers alone do not allow us to draw a parallel with the quality of the test suite in general. Hence, we also calculate the ratio of projects that have the Throw Statement Line Coverage ratio greater than or equal to 60% to enable comparison with the Line Coverage metric results.

As can be observed in Figure 12, we found that six (*i.e.*, two out of 26 (7.69%) desktop/server, one out of five (20%) mobile, and three out of eight (37.50%) multi-platform projects) out of 39 (15.38%) have the Throw Statement Line Coverage ratio greater than or equal to 60%. This is the opposite of what is observed with the Line Coverage metric, in which 74.36% of projects achieve at least 60% of Line Coverage. When considering the domain, 15.38% of frameworks, 27.27% of libraries, and 6.67% of tools achieve the Throw Statement Line Coverage ratio greater than or equal to 60%.

Moreover, if we raise the reference coverage ratio to at least 80%—considered a good coverage to release a piece of software [12, 17]—, we still have 16 out of 39 (41.03%) projects with a Line Coverage ratio greater than or equal to 80%. However, only four (10.26%) projects achieve this coverage when considering the Throw Statement Line Coverage.

Among all projects, we notice in Figure 12 that only one (*i.e.*, ReactiveNetwork) achieved a higher Throw Statement Line Coverage than the Line Coverage of the project.

Notice that 29 out of 39 projects (74.36%) report Line Coverage ratios greater than or equal to 60%. Nevertheless, only six projects achieve a coverage of `throw` statement lines greater than or equal to 60%. Also, even though the mobile platform has a higher median value

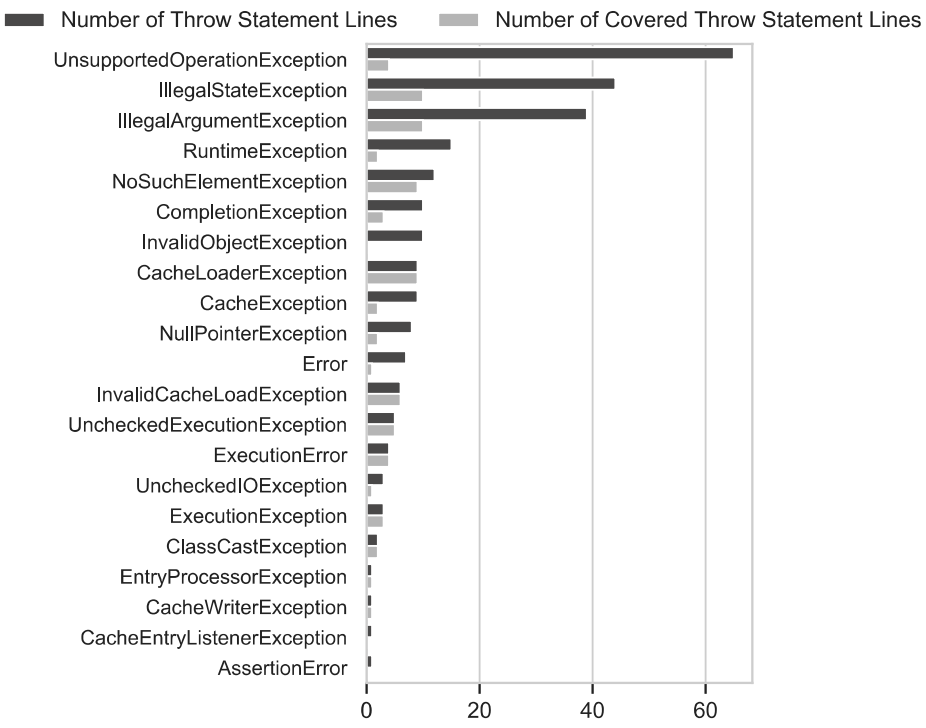


Figure 10 – Caffeine's throw statements data.

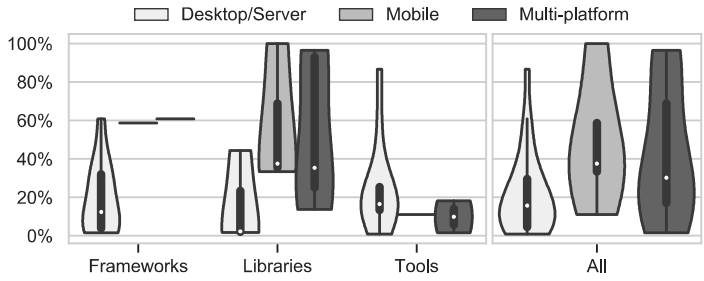


Figure 11 – Throw Statement Line Coverage (NCTS/NTS).

than the other platforms, a larger percentage of multi-platform projects achieve at least 60% of Throw Statement Line Coverage. The libraries showed better results than the other domains once again. These results are in line with the results found and discussed in **RQ1**, and also indicate that the test suites do not exercise the majority of the `throw` statements, even in projects where there seem to have concerns about code coverage. This contributes to the claim that developers tend to neglect the exceptional behavior testing.

RQ4: Do the test suites cover more `throw` statement lines of custom exceptions or standard/third-party exceptions?

To answer this question, we take the Number of Covered Throw Statement Lines of Custom Exceptions (NCTSCE) and divide by the Number of Throw Statement Lines of Custom Exceptions (NTSCE). Likewise, we divide the Number of Covered Throw Statement Lines of

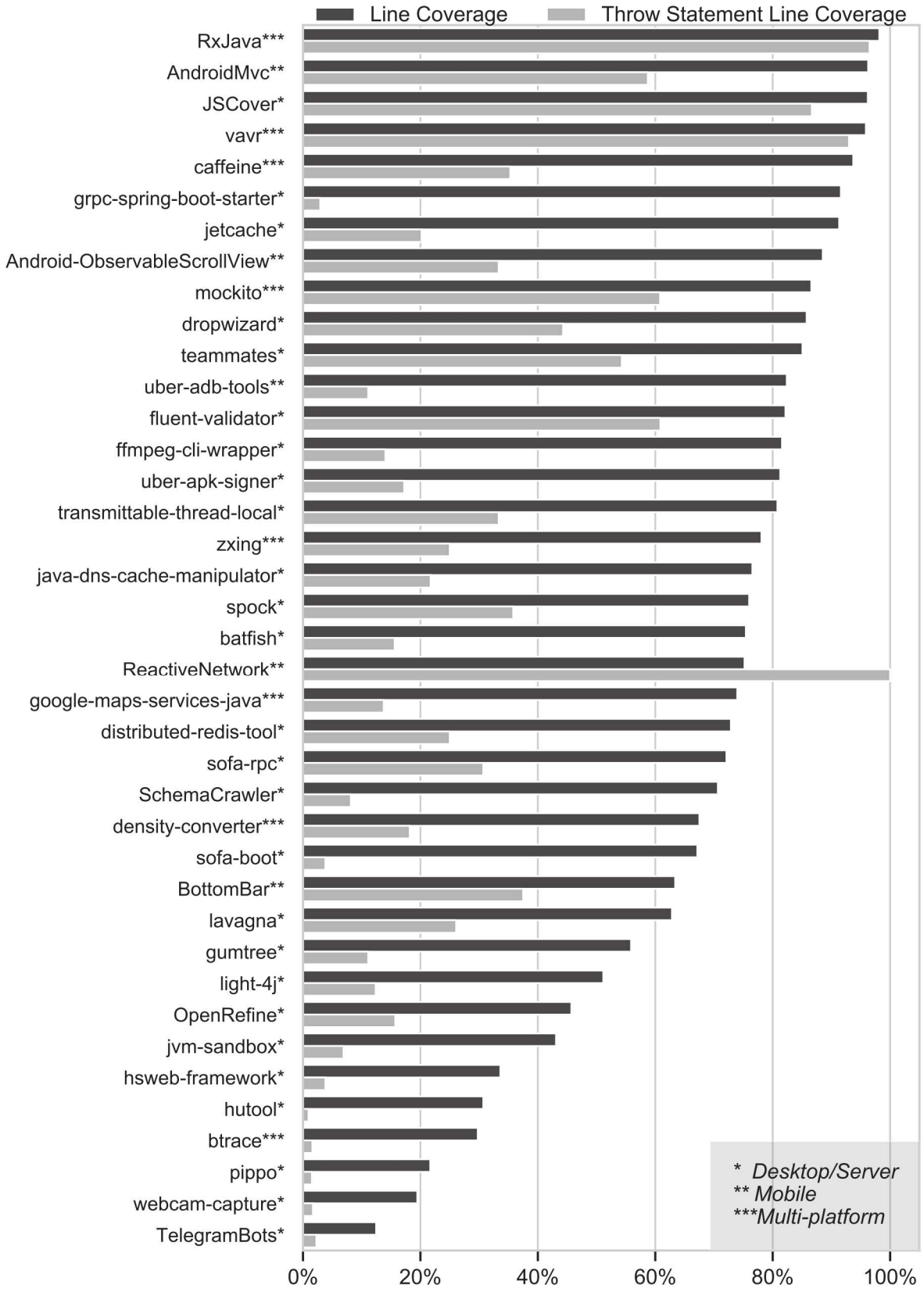
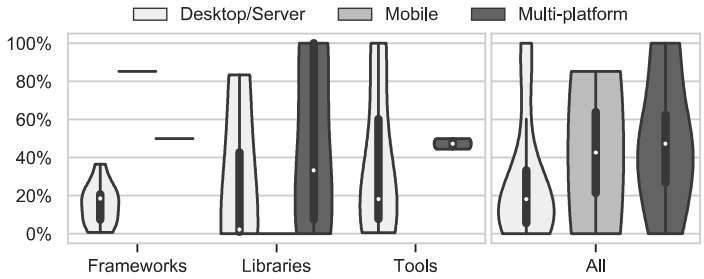


Figure 12 – Coverage Ratios.

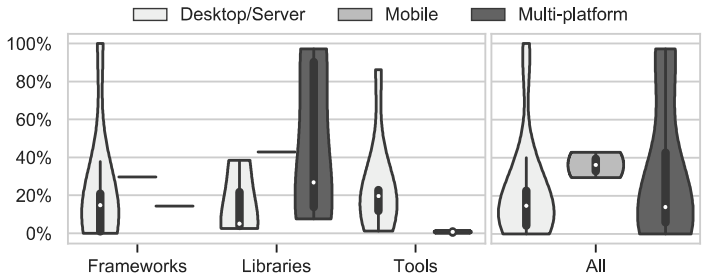
Standard/Third-party Exceptions (NCTSSTPE) by the Number of Throw Statement Lines of

Standard/Third-party Exceptions (NTSSTPE). In this case, we also discarded eight projects that lead to a division by zero in columns NCTSCE/NTSCE or NCTSSTPE/NTSSTPE, and the final number of projects in each platform and domain is presented at the row "count" in Table 7. Figure 13 illustrates the distribution of these ratios.

According to our results, in 16 out of 31 (51.61%) projects the tests cover more `throw` statements of custom exceptions than standard/third-party. Nevertheless, when analyzing by platform, the results are in favor of custom exceptions only in the multi-platform projects (5 out of 8, *i.e.*, 62.50%). The opposite happens in the desktop/server platform (10 out of 21, *i.e.*, 47.62%), and, once again, we have a tie in the mobile platform (1 out of 2, *i.e.*, 50%). Although the difference identified in favor of custom exceptions is minimal, the results are still in line with the **RQ2**'s results.



(a) Custom Exceptions (NCTSCE/NTSCE).



(b) Standard/Third-party Exceptions (NCTSSTPE/NTSSTPE).

Figure 13 – Ratios of NCTSCE/NTSCE and NCTSSTPE/NTSSTPE.

5.3 Exception-testing Constructs Usage Statistics

In this section, we briefly discuss what are the most widely used constructions that we found in the projects analyzed by our tool.

RQ5: How do developers test the exceptional behavior in terms of exception-testing constructs?

To answer this question, we take three JUnit exception-testing constructs into account: `assertThrows` call, the `expected` attribute of the `@Test` annotation, and the

`ExpectedException` rule; four AssertJ exception-testing constructs: `assertThatExceptionName`, `assertThatExceptionOfType`, `assertThat`, and `assertThatThrownBy` calls; one TestNG construct: `expectedExceptions` attribute; and one additional construct common to JUnit, AsssertJ, and TestNG: a `fail` call right before a `catch` block.

As shown in Figure 14, the `fail` method call is the oldest and by far the most used construct: we found 33,100 instances of this construct (56.10% of the overall constructs usage). The `fail` method call is followed by the `expected` attribute (15.42%) of the JUnit Framework and the `assertThatExceptionOfType` call (7.54%) of the AssertJ Library. Notice that the sum of each exception-testing construct is not necessarily equal to the total number of exceptional behavior test methods of the project. This happens because one exceptional behavior test method may have one or more exception-testing constructs even from different frameworks and libraries.

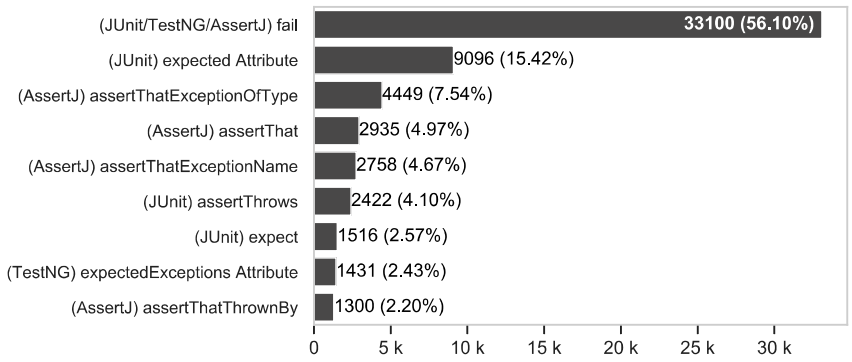


Figure 14 – Number of Exception-testing Constructs.

These numbers also indicate that developers are using generic constructs (*i.e.,* `fail` call) instead of constructs specifically designed to test the exceptional behavior. Moreover, even when we only consider projects up to three years old, we notice that among 55 projects the `fail` construct is still the most used, as shown in Figure 15. These results are important to show that the newer constructs designed to test the exceptional behavior created over the last years should be better spread throughout the developers community so that developers can make the task of testing exceptions less difficult and more efficient.

5.4 Survey

In this section, we discuss the results of our survey with the developers of the analyzed projects.

RQ6: How do developers perceive the exceptional behavior testing?

Overall, 66 developers completed our survey. Figure 16 summarizes the survey results. The majority of the respondents (69.70%) considers exceptional behavior testing as important.

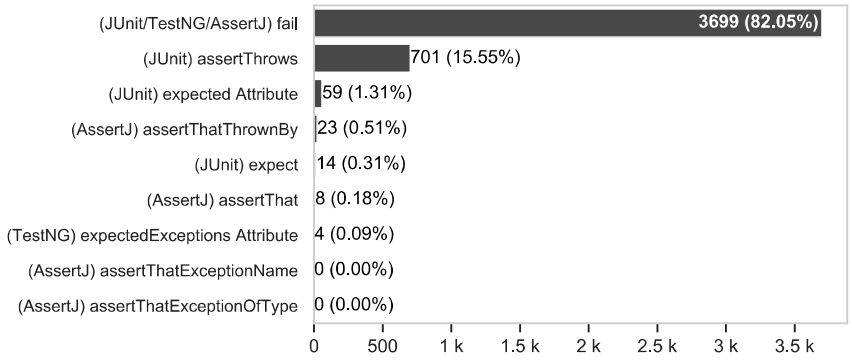
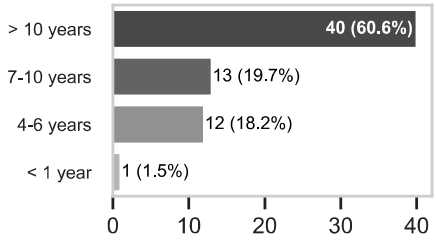
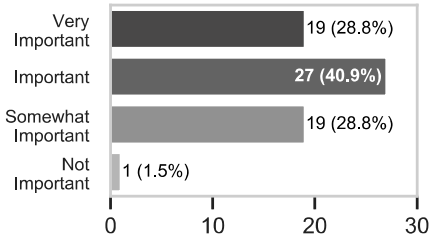


Figure 15 – Number of Exception-testing Constructs found in projects up to three years old.

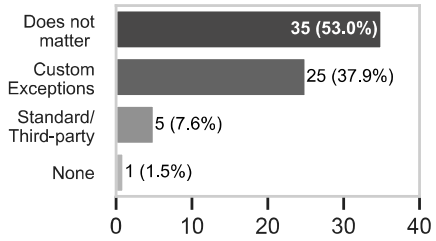
Moreover, 37.90% of the participants prioritize custom exceptions over standard/third-party, which is also in accordance to the findings of **RQ2** and **RQ4**, in particular when considering desktop/server and multi-platform projects.



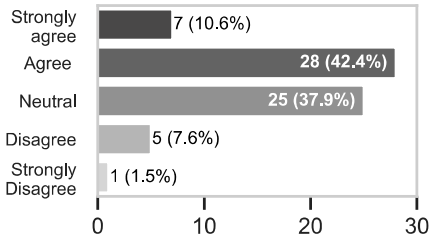
(a) Java experience.



(b) Exceptional tests importance.



(c) Prioritization.



(d) “Developers tend to neglect.”

Figure 16 – 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** and **RQ3**. 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 17 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.”

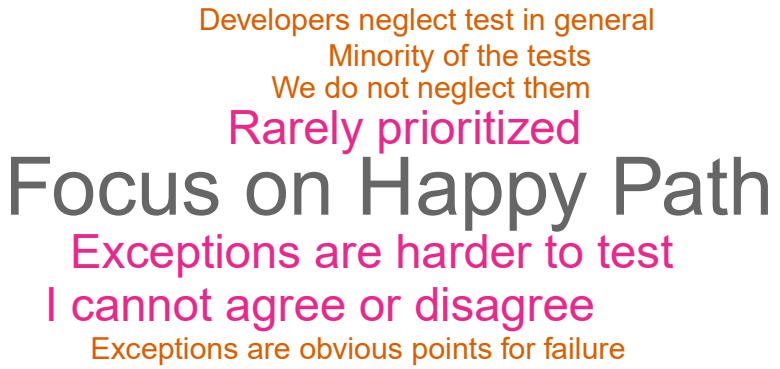


Figure 17 – Word Cloud presenting 8 categories of the comments.

"*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.*"

6 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 [32]. 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.

In our dynamic analysis, the use of Coveralls and Codecov web services to retrieve coverage data is a threat, since developers using these tools may be more concerned with code coverage than those who do not. Also, we do not know what tools were used to calculate coverage and not even if the whole project has coverage data. Moreover, the most used coverage tools report inaccurate behaviors for code snippets related to exceptions. Finally, the segregation of only 39 projects into platforms and domains leads us to draw conclusions from subsets with a small number of projects.

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 article. However, the participants of our survey reported they have great experience in Java and in Java testing frameworks.

7 RELATED WORK

Previous works aim to extend the coverage of testing for exception-handling constructs. Goffi *et al.* [16] 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 [16, 3] is not to analyze whether and to what extent 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.

Sinha et al. [34] used static analysis to detect occurrences of inappropriate coding patterns for exception handling, such as unreachable catch handlers and ignored exceptions, the distance between throw and catch, and imprecise throws declarations. The approach guides the testers in computing test requirements for exception handling and generating test data to satisfy those requirements. Romano et al. [27] use 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.* [15] developed a compiler-directed fault injection static analysis to support white-box coverage testing of exception handlers. They improve coverage of exception handlers through compiler-generated code instrumentation, which can guide the fault injection and record the code exercised by the tests. Martins *et al.* [21] presented *VerifyEx*, a testing tool that uses a source code instrumentation technique to exercise exception handling constructs to increase the coverage rate when testing exceptional behavior. Cornu *et al.* [7] proposed an algorithm that injects exceptions during test suite execution to simulate unanticipated errors. These works provide tools to automatically or semi-automatically improve the generation of tests for exception handling constructs. We also provide a tool. However, the purpose is very different: our tool is able to collect metrics regarding not only exception-handling constructs but also exception-testing ones.

Shah *et al.* [30] conducted a study to understand how developers perceive exception han-

dling and what methods they adopt to deal with exception handling constructs. They interviewed developers and found that they tend to use exception-handling constructs mostly for debugging purposes, which means that the time invested in implementing code for proper handling of error conditions is neglected. Despite not providing results from interviews, we conducted a survey with 66 developers to understand how they perceive the exceptional behavior testing.

Osman *et al.* [25] performed an analysis on 90 Java projects divided into six domains in different versions of each project. They evaluated how developers use the different types of exceptions (standard, custom, and third-party) in throw statements and exception handlers. They observed that applications have significantly more error handling code than libraries, and applications increasingly rely on custom exceptions. Also, projects that belong to different domains have different preferences of exception types. For instance, content management systems rely more on custom exceptions than standard exceptions whereas the opposite is true in parsing frameworks. We also study custom and standard/third-party exceptions, but our effort was on checking whether the tests cover more custom exceptions or standard/third-party ones.

Dúlaigh *et al.* [11] performed an empirical analysis regarding the evolution of twelve Java projects from the *Qualitas Corpus* [35]. They measured the quantity and distribution of exception-handling constructs (`throw` instructions, `throws` clause, and `catch` blocks) and identified that 21.5% of the methods contain at least one exception-handling construct. Differently from our work, they did not collect metrics related to the test of exceptions.

8 CONCLUDING REMARKS

In this work, we presented a *mixed-method* study consisting of: (1) an empirical investigation based on the static analysis of 417 projects and dynamic analysis of 39 projects to understand whether and to what extent developers actually test the exceptional behavior; and (2) a survey with 66 developers from these projects to triangulate our quantitative results.

In our static analysis [8], 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.

Also, we carried out a dynamic analysis on a subset of 39 projects, from our 417 studied projects, with publicly available line coverage reports. The results of our dynamic analysis showed that 29 out of 39 projects (74.36%) report Line Coverage ratios greater than or equal to 60%. Nevertheless, only six (15.38%) projects achieve a coverage of `throw` statement lines greater than or equal to 60%. We observed a similar result when we raised the reference ratio to at least 80%. Thus, the test suites do not exercise the majority of the `throw` statements, even in projects where there seem to have concerns about code coverage. We also observed that a larger percentage of multi-platform projects and libraries have higher numbers to Throw Statement Line Coverage ratios. Our dynamic analysis also provided additional evidence that the custom exceptions have better coverage than the standard/third-party exceptions. Thus, these results are also in accordance to our static analysis results [8].

We also noted that even in projects created over the last years the newer constructs designed to test the exceptional behavior should be better spread throughout the developers community.

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 reinforcement of the importance of the exceptional behavior, the dissemination of good practices for creating exceptional behavior tests, and the use of automatic test suite generation tools.

As future work, we intend to increase and deepen the analysis of projects with coverage data publicly available to better understand the factors that lead to a piece of software with good coverage of the exceptional behavior.

BIBLIOGRAPHY

[1] ANDERSON, T. W., AND FINN, J. D. *The new statistical analysis of data.* Springer, 1996.

[2] ASADUZZAMAN, M., AHASANUZZAMAN, M., ROY, C. K., AND SCHNEIDER, K. A. How developers use exception handling in Java? In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), MSR '16, ACM Press, pp. 516–519.

[3] BERNARDO, R. D., JR., R. S., CASTOR, F., COELHO, R., CACHO, N., AND SOARES, S. Agile testing of exceptional behavior. In *Proceedings of the 25th Brazilian Symposium on Software Engineering* (2011), SBES '11, pp. 204–213.

[4] CABRAL, B., AND MARQUES, P. Exception handling: A field study in Java and .NET. In *Proceedings of the 21st European Conference on Object-Oriented Programming* (2007), ECOOP '07, Springer, pp. 151–175.

[5] CASSEE, N., PINTO, G., CASTOR, F., AND SEREBRENIK, A. How Swift developers handle errors. In *Proceedings of the 15th International Conference on Mining Software Repositories* (2018), MSR '15, ACM Press, pp. 292–302.

[6] CHANG, B.-M., AND CHOI, K. A review on exception analysis. *Information and Software Technology 77*, C (sep 2016), 1–16.

[7] CORNU, B., SEINTURIER, L., AND MONPERRUS, M. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology 57* (2015), 66 – 76.

[8] DALTON, F., RIBEIRO, M., PINTO, G., FERNANDES, L., GHEYI, R., AND FONSECA, B. Is exceptional behavior testing an exception? an empirical assessment using Java automated tests. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering* (2020), EASE '20, ACM Press, p. 170–179.

[9] DE PÁDUA, G. B., AND SHANG, W. Studying the prevalence of exception handling anti-patterns. In *Proceeedings of the 25th International Conference on Program Comprehension* (2017), ICPC '17, pp. 328–331.

[10] DE PÁDUA, G. B., AND SHANG, W. Studying the relationship between exception handling practices and post-release defects. In *Proceedings of the 15th International Conference on Mining Software Repositories* (2018), MSR '18, ACM Press, pp. 564–575.

[11] DÚLAIGH, K. O., POWER, J. F., AND CLARKE, P. J. Measurement of exception-handling code: An exploratory study. In *Proceedings of the 5th International Workshop on Exception Handling* (2012), WEH '12, IEEE Press, pp. 55–61.

[12] EMMA TEAM. What is code coverage and why should I care about it?, 2006. (Accessed May 2020).

[13] ENGINEERING AND SYSTEMS SOFTWARE RESEARCH GROUP (EASY). Research replication package, 2020. (Accessed May 2020).

[14] FAN, L., SU, T., CHEN, S., MENG, G., LIU, Y., XU, L., PU, G., AND SU, Z. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering* (2018), ICSE '18, ACM Press, pp. 408–419.

[15] FU, C., MILANOVA, A. L., G, B., RYDER, AND WONNACOTT, D. G. Robustness testing of Java server applications. *IEEE Transactions on Software Engineering 31*, 4 (2005), 292–311.

[16] GOFFI, A., GORLA, A., ERNST, M. D., AND PEZZÈ, M. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), ISSTA '16, pp. 213–224.

[17] GOPINATH, R., JENSEN, C., AND GROCE, A. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ICSE' 2014, ACM Press, pp. 72—-82.

[18] JIANG, S., ZHANG, Y., YAN, D., AND JIANG, Y. An approach to automatic testing exception handling. *SIGPLAN Not. 40*, 8 (Aug. 2005), 34—-39.

[19] MAO, C.-Y., AND LU, Y.-S. 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* (2005), ICECCS '05, IEEE Press, pp. 432–439.

[20] MARICK, B. How to misuse code coverage. In *Proceedings of the 16th International Conference on Testing Computer Software* (1999).

[21] MARTINS, A. L., HANAZUMI, S., AND DE MELO, A. C. Testing Java exceptions: An instrumentation technique. In *IEEE 38th International Computer Software and Applications Conference Workshops* (2014), COMPSACW '14, pp. 626–631.

[22] MELO, H., COELHO, R., AND TREUDE, C. Unveiling exception handling guidelines adopted by Java developers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering* (2019), SANER '19, pp. 128–139.

[23] MONTENEGRO, T., MELO, H., COELHO, R., AND BARBOSA, E. Improving developers awareness of the exception handling policy. In *Proceedings of the 25th International*

Conference on Software Analysis, Evolution and Reengineering (2018), SANER '18, pp. 413–422.

[24] MYERS, G. J., SANDLER, C., AND BADGETT, T. *The Art of Software Testing*, 3rd ed. Wiley Publishing, 2011.

[25] OSMAN, H., CHIS, A., CORRODI, C., GHAFARI, M., AND NIERSTRASZ, O. Exception evolution in long-lived Java systems. In *Proceedings of the 14th International Conference on Mining Software Repositories* (2017), MSR '17, IEEE Press, pp. 302–311.

[26] REIMER, D., AND SRINIVASAN, H. Analyzing exception usage in large java applications. In *Workshop on Exception Handling in Object Oriented Systems* (2003), EHOOS '03.

[27] ROMANO, D., PENTA, M. D., AND ANTONIOL, G. An approach for search based testing of null pointer exceptions. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation* (2011), ICST '11, IEEE Press, pp. 160–169.

[28] RYDER, B. G., SMITH, D., KREMER, U. J., GORDON, M. D., AND SHAH, N. A static study of Java exceptions using JESP. In *Proceedings of the 9th International Conference on Compiler Construction* (2000), CC '00, Springer, pp. 67–81.

[29] SENA, D., COELHO, R., KULESZA, U., AND BONIFÁCIO, R. Understanding the exception handling strategies of Java libraries: An empirical study. In *Proceedings of the 13th Working Conference on Mining Software Repositories* (2016), MSR '16, ACM Press, pp. 212–222.

[30] SHAH, H., GÖRG, C., AND HARROLD, M. J. Why do developers neglect exception handling? In *Proceedings of the 4th International Workshop on Exception Handling (WEH)* (2008), ACM, pp. 62–68.

[31] SHAPIRO, S. S., AND MARTIN, B. W. An analysis of variance test for normality. *Biometrika 52* (1965), 591–611.

[32] SILVA, H., AND VALENTE, M. T. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software 146* (2018), 112–129.

[33] SINHA, S., AND HARROLD, M. J. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering 26*, 9 (2000), 849–871.

[34] SINHA, S., ORSO, A., AND HARROLD, M. J. Automated support for development, maintenance, and testing in the presence of implicit flow control. In *in Proceedings of the 26th International Conference on Software Engineering (ICSE)* (2004).

[35] TEMPERO, E., ANSLOW, C., DIETRICH, J., HAN, T., LI, J., LUMPE, M., MELTON, H., AND NOBLE, J. The qualitas corpus: A curated collection of Java code for empirical studies. In *in Proceedings of the Asia-Pacific Software Engineering Conference (ASPEC)*.

[36] WILLIAMS, T., MERCER, M., MUCHA, J., AND KAPUR, R. Code coverage, what does it mean in terms of quality? pp. 420–424.

[37] WIRFS-BROCK, R. J. Toward exception-handling best practices and patterns. *IEEE Press 23*, 5 (2006), 11–13.

[38] ZHANG, P., AND ELBAUM, S. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *Transactions on Software Engineering and Methodology 23*, 4 (2014), 32:1–32:28.

[39] ZHU, H., HALL, P. A. V., AND MAY, J. H. R. Software unit test coverage and adequacy. *ACM Computing Surveys 29*, 4 (Dec. 1997), 366—427.

APPENDIX A – TABLES

Table 2 – 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**

Table 3 – 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.*

		NDUCE	NDUSTE	NDUE	NDUCE/ NDUE	NDUSTE/ NDUE	
Desktop/Server	Frameworks	78	78	78	78	78	**count**
		15.36	35.56	50.92	22.12%	77.88%	**mean**
		22.94	25.67	46.68	16.25%	16.25%	**std**
		0	2	3	0%	33.33%	**min**
		5	29	37	18.98%	81.02%	**50%**
		138	133	252	66.67%	100%	**max**
	Libraries	34	34	34	34	34	**count**
		14.65	41.56	56.21	17.91%	82.09%	**mean**
		27.58	52.94	78.92	13.08%	13.08%	**std**
		0	1	1	0%	42.48%	**min**
		4.50	30.50	34.50	17.03%	82.97%	**50%**
		143	317	460	57.52%	100%	**max**
	Tools	90	90	90	90	90	**count**
		8.82	29.97	38.79	12.95%	87.05%	**mean**
		25.64	22.92	38.89	14.82%	14.82%	**std**
		0	3	3	0%	11.28%	**min**
		2.50	26	27.50	8.86%	91.14%	**50%**
		228	135	257	88.72%	100%	**max**
	All	202	202	202	202	202	**count**
		12.33	34.08	46.41	17.33%	82.67%	**mean**
		25.05	31.03	50.77	15.62%	15.62%	**std**
		0	1	1	0%	11.28%	**min**
		4	27	34	14.82%	85.18%	**50%**
		228	317	460	88.72%	100%	**max**
Mobile	Frameworks	50	50	50	50	50	**count**
		8.40	23.16	31.56	12.01%	87.99%	**mean**
		28.57	30.19	55.89	14.33%	14.33%	**std**
		0	1	1	0%	45.40%	**min**
		1	11	12	6.70%	93.30%	**50%**
		184	153	337	54.60%	100%	**max**
	Libraries	60	60	60	60	60	**count**
		2.02	12.17	14.18	6.43%	93.57%	**mean**
		5.87	13.86	19.31	9.11%	9.11%	**std**
		0	1	1	0%	67.69%	**min**
		0	8	9	0%	100%	**50%**
		42	88	130	32.31%	100%	**max**
	Tools	42	42	42	42	42	**count**
		1.40	15.10	16.50	3.93%	96.07%	**mean**
		3.31	12.32	14.86	7.30%	7.30%	**std**
		0	1	1	0%	73.33%	**min**
		0	12	12	0%	100%	**50%**
		16	51	60	26.67%	100%	**max**
	All	152	152	152	152	152	**count**
		3.95	16.59	20.54	7.58%	92.42%	**mean**
		17.06	20.85	35.78	11.15%	11.15%	**std**
		0	1	1	0%	45.40%	**min**
		0	9	10	0%	100%	**50%**
		184	153	337	54.60%	100%	**max**
Multi-platform	Frameworks	22	22	22	22	22	**count**
		22.18	46	68.18	21.37%	78.63%	**mean**
		42.68	39.17	78.92	15.19%	15.19%	**std**
		0	4	4	0%	45.45%	**min**
		11	35	46	23.53%	76.47%	**50%**
		204	170	374	54.55%	100%	**max**
	Libraries	35	35	35	35	35	**count**
		6.86	26.57	33.43	12.23%	87.77%	**mean**
		12.88	17.92	28.30	13.16%	13.16%	**std**
		0	2	2	0%	50%	**min**
		2	24	26	10%	90%	**50%**
		61	78	122	50%	100%	**max**
	Tools	6	6	6	6	6	**count**
		8.33	30.67	39	19.88%	80.12%	**mean**
		10.41	28.01	37.68	11.35%	11.35%	**std**
		1	7	11	6.98%	63.64%	**min**
		3.50	23.50	27.50	20.84%	79.16%	**50%**
		28	80	108	36.36%	93.02%	**max**
	All	63	63	63	63	63	**count**
		12.35	33.75	46.10	16.15%	83.85%	**mean**
		27.74	29.02	54.14	14.25%	14.25%	**std**
		0	2	2	0%	45.45%	**min**
		3	28	36	13.33%	86.67%	**50%**
		204	170	374	54.55%	100%	**max**

Table 4 – Metrics to answer **RQ2** (continued). ***NDTCE***=*Number of Distinct Tested Custom Exceptions;* ***NDTSTE***=*Number of Distinct Tested Standard/Third-party Exceptions;* ***NDTE***=*Number of Distinct Tested Exceptions.*

		NDTCE	NDTSTE	NDTE	NDTCE/ NDTE	NDTSTE/ NDTE	
Desktop/Server	Frameworks	78	78	78	78	78	**count**
		4.03	4.88	8.91	35.44%	64.56%	**mean**
		7.53	7.03	13.94	27.98%	27.98%	**std**
		0	0	0	0%	0%	**min**
		1	2	3	38.60%	61.40%	**50%**
		34	34	65	100%	100%	**max**
	Libraries	34	34	34	34	34	**count**
		4.18	6.35	10.53	26.88%	73.12%	**mean**
		9.68	7.16	15.79	22.24%	22.24%	**std**
		0	0	0	0%	22.22%	**min**
		1	4.50	5.50	27.27%	72.73%	**50%**
		56	30	86	77.78%	100%	**max**
	Tools	90	90	90	90	90	**count**
		2.37	2.40	4.77	32.67%	67.33%	**mean**
		11.25	3.58	12.53	36.29%	36.29%	**std**
		0	0	0	0%	0%	**min**
		0	1	1	26.79%	73.22%	**50%**
		104	17	109	100%	100%	**max**
	All	202	202	202	145	145	**count**
		3.31	4.02	7.34	32.58%	67.42%	**mean**
		9.69	5.95	13.80	30.73%	30.73%	**std**
		0	0	0	0%	0%	**min**
		0	2	2	30.43%	69.57%	**50%**
		104	34	109	100%	100%	**max**
Mobile	Frameworks	50	50	50	17	17	**count**
		1.22	2.98	4.20	22.38%	77.62%	**mean**
		3.82	7.25	10.61	28.90%	28.90%	**std**
		0	0	0	0%	0%	**min**
		0	0	0	10%	90%	**50%**
		22	36	58	100%	100%	**max**
	Libraries	60	60	60	24	24	**count**
		0.33	1.45	1.78	13.13%	86.87%	**mean**
		1.07	2.75	3.52	23.68%	23.68%	**std**
		0	0	0	0%	0%	**min**
		0	0	0	0%	100%	**50%**
		6	13	14	100%	100%	**max**
	Tools	42	42	42	7	7	**count**
		0.07	0.33	0.40	16.67%	83.33%	**mean**
		0.26	0.93	1.11	23.57%	23.57%	**std**
		0	0	0	0%	50%	**min**
		0	0	0	0%	100%	**50%**
		1	5	6	50%	100%	**max**
	All	152	152	152	48	48	**count**
		0.55	1.64	2.20	16.92%	83.08%	**mean**
		2.33	4.62	6.63	25.45%	25.45%	**std**
		0	0	0	0%	0%	**min**
		0	0	0	0%	100%	**50%**
		22	36	58	100%	100%	**max**
Multi-platform	Frameworks	22	22	22	18	18	**count**
		10.77	10.05	20.82	36.74%	63.26%	**mean**
		25.19	11.81	35.35	30.59%	30.59%	**std**
		0	0	0	0%	0%	**min**
		2	7.50	9.50	34.52%	65.48%	**50%**
		116	44	160	100%	100%	**max**
	Libraries	35	35	35	29	29	**count**
		2.23	6.54	8.77	25.39%	74.61%	**mean**
		3.39	6.80	9.36	24.95%	24.95%	**std**
		0	0	0	0%	0%	**min**
		1	5	7	18.18%	81.82%	**50%**
		14	29	41	100%	100%	**max**
	Tools	6	6	6	5	5	**count**
		1.67	1.67	3.33	50.77%	49.23%	**mean**
		2.73	2.42	4.84	50.03%	50.03%	**std**
		0	0	0	0%	0%	**min**
		0.50	0.50	1.50	53.85%	46.15%	**50%**
		7	6	13	100%	100%	**max**
	All	63	63	63	52	52	**count**
		5.16	7.30	12.46	31.76%	68.24%	**mean**
		15.46	8.89	22.66	30.28%	30.28%	**std**
		0	0	0	0%	0%	**min**
		1	4	7	22.50%	77.50%	**50%**
		116	44	160	100%	100%	**max**

Table 5 – Metrics to answer **RQ2** (continued). ***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.*

		NDTCE/NDUCE	NDTSTE/NDUSTE	
Desktop/Server	Frameworks	68	68	**count**
		26.18%	12.09%	**mean**
		27.69%	13.17%	**std**
		0%	0%	**min**
		20.84%	7.90%	**50%**
		100%	53.12%	**max**
	Libraries	31	31	**count**
		34.16%	17.31%	**mean**
		31.58%	15.32%	**std**
		0%	0%	**min**
		33.33%	15.38%	**50%**
		100%	70%	**max**
	Tools	64	64	**count**
		28.75%	8.14%	**mean**
		36.93%	9.40%	**std**
		0%	0%	**min**
		11.80%	3.85%	**50%**
		100%	31.71%	**max**
	All	163	163	**count**
		28.71%	11.53%	**mean**
		32.26%	12.67%	**std**
		0%	0%	**min**
		20%	7.69%	**50%**
		100%	70%	**max**
Mobile	Frameworks	30	30	**count**
		10.33%	9.35%	**mean**
		20.57%	14.12%	**std**
		0%	0%	**min**
		0%	0%	**50%**
		66.67%	55.56%	**max**
	Libraries	24	24	**count**
		14.59%	12.20%	**mean**
		26.20%	14.89%	**std**
		0%	0%	**min**
		0%	9.09%	**50%**
		100%	53.33%	**max**
	Tools	12	12	**count**
		12.78%	3.52%	**mean**
		29.47%	9.01%	**std**
		0%	0%	**min**
		0%	0%	**50%**
		100%	31.25%	**max**
	All	66	66	**count**
		12.33%	9.33%	**mean**
		24.13%	13.80%	**std**
		0%	0%	**min**
		0%	0%	**50%**
		100%	55.56%	**max**
Multi-platform	Frameworks	18	18	**count**
		42.36%	21.46%	**mean**
		29.61%	16.37%	**std**
		0%	0%	**min**
		50%	25.90%	**50%**
		100%	57.14%	**max**
	Libraries	26	26	**count**
		52.26%	26.53%	**mean**
		34.22%	20.70%	**std**
		0%	0%	**min**
		50%	19.44%	**50%**
		100%	78.95%	**max**
	Tools	6	6	**count**
		26.39%	3.82%	**mean**
		29.07%	6.49%	**std**
		0%	0%	**min**
		25%	1.25%	**50%**
		58.33%	16.67%	**max**
	All	50	50	**count**
		45.59%	21.98%	**mean**
		32.57%	19.17%	**std**
		0%	0%	**min**
		50%	17.80%	**50%**
		100%	78.95%	**max**

Table 6 – Metrics to answer **RQ3**. ***NCTS***=*Number of Covered Throw Statement Lines;* ***NTS***=*Number of Throw Statement Lines.*

		NCTS	NTS	Throw Statement Line Coverage (NCTS/NTS)	Line Coverage	
Desktop/Server	Frameworks	11	11	11	11	**count**
		26.55	144.09	19.30%	64.61%	**mean**
		39.82	117.87	18.96%	23.78%	**std**
		1	12	1.52%	21.68%	**min**
		8	105	12.38%	72.12%	**50%**
		121	394	60.87%	91.61%	**max**
	Libraries	3	3	3	3	**count**
		28.33	292.33	16.09%	39.24%	**mean**
		36.25	168.94	24.43%	40.46%	**std**
		4	158	1.69%	12.46%	**min**
		11	237	2.28%	19.47%	**50%**
		70	482	44.30%	85.78%	**max**
	Tools	12	12	12	12	**count**
		35.50	246.50	24.72%	69.57%	**mean**
		57.07	376.86	23.52%	18.19%	**std**
		1	4	0.94%	30.71%	**min**
		8	51	16.48%	74.16%	**50%**
		184	1,179	86.67%	96.20%	**max**
	All	26	26	26	26	**count**
		30.88	208.46	21.43%	63.97%	**mean**
		46.82	271.46	21.10%	24.31%	**std**
		1	4	0.94%	12.46%	**min**
		8	131.50	15.67%	72.50%	**50%**
		184	1,179	86.67%	96.20%	**max**
Mobile	Frameworks	1	1	1	1	**count**
		37	63	58.73%	96.26%	**mean**
		-	-	-	-	**std**
		37	63	58.73%	96.26%	**min**
		37	63	58.73%	96.26%	**50%**
		37	63	58.73%	96.26%	**max**
	Libraries	3	3	3	3	**count**
		2.67	5	56.94%	75.71%	**mean**
		1.53	2.65	37.35%	12.58%	**std**
		1	3	33.33%	63.39%	**min**
		3	4	37.50%	75.20%	**50%**
		4	8	100%	88.53%	**max**
	Tools	1	1	1	1	**count**
		2	18	11.11%	82.40%	**mean**
		-	-	-	-	**std**
		2	18	11.11%	82.40%	**min**
		2	18	11.11%	82.40%	**50%**
		2	18	11.11%	82.40%	**max**
	All	5	5	5	5	**count**
		9.40	19.20	48.13%	81.16%	**mean**
		15.47	25.19	33.56%	12.60%	**std**
		1	3	11.11%	63.39%	**min**
		3	8	37.50%	82.40%	**50%**
		37	63	100%	96.26%	**max**
Multi-platform	Frameworks	1	1	1	1	**count**
		149	245	60.82%	86.58%	**mean**
		-	-	-	-	**std**
		149	245	60.82%	86.58%	**min**
		149	245	60.82%	86.58%	**50%**
		149	245	60.82%	86.58%	**max**
	Libraries	5	5	5	5	**count**
		162	290.20	52.71%	87.96%	**mean**
		124.17	140.28	39.14%	11.11%	**std**
		11	80	13.75%	73.96%	**min**
		118	286	35.31%	93.72%	**50%**
		305	472	96.49%	98.16%	**max**
	Tools	2	2	2	2	**count**
		3.50	104.50	9.89%	48.64%	**mean**
		0.71	116.67	11.72%	26.69%	**std**
		3	22	1.60%	29.77%	**min**
		3.50	104.50	9.89%	48.64%	**50%**
		4	187	18.18%	67.52%	**max**
	All	8	8	8	8	**count**
		120.75	238.12	43.02%	77.96%	**mean**
		118.61	142.25	36.34%	22.36%	**std**
		3	22	1.60%	29.77%	**min**
		109.50	265	30.16%	82.33%	**50%**
		305	472	96.49%	98.16%	**max**

Table 7 – Metrics to answer **RQ4**. ***NCTSCE***=*Number of Covered Throw Statement Lines of Custom Exceptions;* ***NTSCE***=*Number of Throw Statement Lines of Custom Exceptions;* ***NCTPSSTPE***=*Number of Covered Throw Statement Lines of Standard/Third-party Exceptions;* ***NTSSTPE***=*Number of Throw Statement Lines of Standard/Third-party Exceptions.*

		NCTSCE	NTSCE	NCTSCE/ NTSCE	NCTSSTPE	NTSSTPE	NCTSSTPE/ NTSSTPE	
Desktop/Server	Frameworks	11	11	9	11	11	9	**count**
		9.82	63.82	15.46%	8.91	56.64	21.91%	**mean**
		14.84	60.46	11.10%	11.55	43.83	31.85%	**std**
		0	0	0.67%	0	9	0%	**min**
		4	52	18.52%	4	44	14.77%	**50%**
		43	169	36.44%	37	149	100%	**max**
	Libraries	3	3	3	3	3	3	**count**
		6.67	182.33	28.50%	16	96.33	15.31%	**mean**
		5.77	243.32	47.50%	23.43	69.83	20.02%	**std**
		0	12	0%	1	20	2.55%	**min**
		10	74	2.17%	4	112	5%	**50%**
		10	461	83.33%	43	157	38.39%	**max**
	Tools	12	12	9	12	12	9	**count**
		21.25	144.92	37.66%	13.17	88.33	24.45%	**mean**
		42.39	248.51	39.68%	19.30	112.12	25.80%	**std**
		0	0	0.56%	1	4	1.15%	**min**
		2	12.50	18.18%	5	28.50	19.61%	**50%**
		112	761	100%	70	357	86.21%	**max**
	All	26	26	21	26	26	21	**count**
		14.73	114.92	26.84%	11.69	75.85	22.06%	**mean**
		30.34	188.41	31.87%	16.36	83.53	26.86%	**std**
		0	0	0%	0	4	0%	**min**
		3.50	44	18.18%	5	43.50	14.77%	**50%**
		112	761	100%	70	357	100%	**max**
Mobile	Frameworks	1	1	1	1	1	1	**count**
		29	34	85.29%	8	27	29.63%	**mean**
		-	-	-	-	-	-	**std**
		29	34	85.29%	8	27	29.63%	**min**
		29	34	85.29%	8	27	29.63%	**50%**
		29	34	85.29%	8	27	29.63%	**max**
	Libraries	3	3	1	3	3	1	**count**
		0	0.33	0%	2.67	4.67	42.86%	**mean**
		0	0.58	-	1.53	2.08	-	**std**
		0	0	0%	1	3	42.86%	**min**
		0	0	0%	3	4	42.86%	**50%**
		0	1	0%	4	7	42.86%	**max**
	Tools	1	1	0	1	1	0	**count**
		0	0	-	2	17	-	**mean**
		-	-	-	-	-	-	**std**
		0	0	-	2	17	-	**min**
		0	0	-	2	17	-	**50%**
		0	0	-	2	17	-	**max**
	All	5	5	2	5	5	2	**count**
		5.80	7	42.64%	3.60	11.60	36.24%	**mean**
		12.97	15.10	60.31%	2.70	10.24	9.36%	**std**
		0	0	0%	1	3	29.63%	**min**
		0	0	42.64%	3	7	36.24%	**50%**
		29	34	85.29%	8	27	42.86%	**max**
Multi-platform	Frameworks	1	1	1	1	1	1	**count**
		43	86	50%	7	49	14.29%	**mean**
		-	-	-	-	-	-	**std**
		43	86	50%	7	49	14.29%	**min**
		43	86	50%	7	49	14.29%	**50%**
		43	86	50%	7	49	14.29%	**max**
	Libraries	5	5	5	5	5	5	**count**
		3.80	17.40	48.14%	93.40	178.80	47.16%	**mean**
		3.42	28.45	48.94%	90.69	67.47	43.15%	**std**
		0	1	0%	10	72	7.60%	**min**
		3	6	33.33%	66	179	26.83%	**50%**
		9	68	100%	204	246	97.21%	**max**
	Tools	2	2	2	2	2	2	**count**
		2.50	5.50	47.22%	1	77	0.71%	**mean**
		2.12	4.95	3.93%	1.41	90.51	1%	**std**
		1	2	44.44%	0	13	0%	**min**
		2.50	5.50	47.22%	1	77	0.71%	**50%**
		4	9	50%	2	141	1.42%	**max**
	All	8	8	8	8	8	8	**count**
		8.38	23	48.14%	59.50	137.12	31.44%	**mean**
		14.26	33.81	37.03%	83.02	84.59	39.40%	**std**
		0	1	0%	0	13	0%	**min**
		3.50	7.50	47.22%	11.50	156	14.09%	**50%**
		43	86	100%	204	246	97.21%	**max**