How Java Programmers Test Exceptional Behavior

Diego Marcilio
USI Università della Svizzera italiana
Lugano, Switzerland
dvmarcilio.github.io

Carlo A. Furia
USI Università della Svizzera italiana
Lugano, Switzerland
bugcounting.net

***Abstract*—Exceptions often signal faulty or undesired behavior; hence, high-quality test suites should also target exceptional behavior. This paper is a large-scale study of *exceptional tests*—which exercise exceptional behavior—in 1 157 open-source Java projects hosted on GitHub. We analyzed JUnit exceptional tests to understand what kinds of exceptions are more frequently tested, what coding patterns are used, and how features of a project, such as its size and number of contributors, correlate to the characteristics of its exceptional tests. We found that exceptional tests are only 13% of all tests, but tend to be larger than other tests on average; unchecked exceptions are tested twice as frequently as checked ones; 42% of all exceptional tests use `try/catch` blocks and usually are larger than those using other idioms; and bigger projects with more contributors tend to have more exceptional tests written using different styles. The paper also zeroes in on several detailed examples involving some of the largest analyzed projects, which refine the empirical results with qualitative evidence. The study's findings, and the capabilities of the tool we developed to analyze exceptional tests, suggest several implications for the practice of software development and for follow-up empirical studies.**

I. Introduction

The importance of testing in software development has become conventional wisdom; yet, writing high-quality tests remains a challenging endeavor [1], [2]. Among all different kinds of tests that are written, in this paper we focus on those that *exercise exceptional behavior*—or *exceptional tests* for short. Exceptional behavior is a frequent source of failures [3], and is often implicated in anti-patterns and misuses [4]; on the other hand, proper exception-handling code is a necessary component of robust, maintainable software [5], [6]. Therefore, testing exceptional behavior is critical in building comprehensive test suites. However, dealing with exceptions—including in tests[a]—can be tricky, because an exceptional behavior's control flow is intrinsically unstructured (an exception can propagate through the call stack) and it is easy to miss some "corner cases" of exception-inducing inputs [7]. As we point out in Sec. III, testing practices have been studied extensively, and exceptional behavior is an increasingly popular empirical research target, but the combination of the two topics—exceptional testing—has so far received little attention.

In this paper, we contribute to narrowing this knowledge gap with a *large-scale empirical study of exceptional testing in Java*. As we describe in Sec. VII, we analyzed all exceptional tests we could detect written using any version of the JUnit framework in 1 157 open-source Java projects—including numerous widely-used frameworks maintained by Apache, Google, and Spring—comprising 1 123 846 tests. The main **findings** of this analysis, detailed in Sec. V, include:

- Exceptional tests are often included as part of the test-writing effort: 66% of projects with tests also include some exceptional tests, and 13% of all tests are exceptional. On average, an exceptional test is 110% the size (mean lines of code) of any test.
- Exceptional tests most frequently target Java's standard exception classes (over $2/3$ of all exceptional tests), and unchecked exceptions (about twice as frequently as checked exceptions).
- A standard **`try/catch`** block is the most common way of writing an exceptional test, followed by JUnit's `@Test(expected=...)` annotation.
- Exceptional tests written using **`try/catch`** blocks tend to be the longest; those written using `@Test(expected=...)` tend to be the shortest.
- Larger projects with more contributors are more likely to include exceptional tests written in a variety of styles.

In addition to several more quantitative findings, Sec. V includes selected **qualitative** evidence about some of the largest projects we analyzed, which complements and hone the numerical data with concrete, illustrative examples.

In summary, this paper makes the following **contributions**:

- *JUnitScrambler*: a tool that automatically discovers and analyzes (exceptional) tests written in any version of JUnit (described in Sec. IV-C).
- The dataset obtained by running *JUnitScrambler* on over a thousand open-source Java projects, including information about over a million tests.[b]
- The empirical analysis of the dataset, described in Sec. IV–V.

The study's findings, and the techniques used for the analysis, suggest several implications for the practice of software development and for follow-up empirical studies. Sec. VII outlines some of them on concrete examples from our analysis.

[a] A StackOverflow question[1] asking how to write exceptional tests in JUnit has over 1.3 million views and several answers from experts such as one of Mockito's core contributors and StackOverflow users with high reputation.

[b] A replication package including tool, dataset, and all scripts is available at https://doi.org/10.6084/m9.figshare.13547561.

II. Background

A. Exceptions: What They Are For

Exceptions are used to signal that something went wrong during program execution. A program may include *exception handling* code, which executes when an exception is raised to try to recover from the error or at least mitigate it. Thus, at a high-level, exceptions can help improve program *robustness*.

In an object-oriented language like Java, exceptions are instances of some *exception classes*; different exception classes characterize different ways of using exceptions. In our analysis, we consider three orthogonal (and standard [6], [8], [9]) classifications according to *origin*, *kind*, and behavioral *usage*.

Origin: an exception class's *origin* in a given project depends on where it is defined: in Java's standard libraries, local to the project's code base, or in an external library.

Kind: according to its type,[2] an exception may be unchecked (a subtype of `RuntimeException` or `Error`) or checked (any other subtype of `Throwable`).[3]

Usage: exceptions are used to signal three main different categories of program behavior [8, §4.4] [9, §12] [6, §8.4], which we refer to as *usage* failure, fault, and return. A failure is a low-level error that usually depends on an exceptional state of the execution environment; for instance, the program runs out of memory (`OutOfMemoryError`). A fault signals the violation of a program's expected behavior; for instance, an array is accessed with an invalid index (`ArrayIndexOutOfBoundsException`). Category return captures improper usages of exceptions—not to signal erroneous conditions but to propagate information outside of the language's structured control flow. For example, to "break out of a complex, nested control flow"[4] similarly to a goto; or to signal the end of a file (`java.io.EOFException`).

B. Exceptional Testing Patterns

An *exceptional test* is a test that may trigger exceptional behavior in the code it exercises. Based on the documentation of JUnit[5] and other testing libraries,[6] as well as on other empirical studies [10]–[13], we identified five main coding *patterns* that programmers use to write exceptional tests—which we show in Fig. 1 and discuss in the rest of this section.

1) Pattern try/catch: Pattern try/catch uses Java's built-in **`try/catch`** statements, and hence it does not require any library. Testing whether some `testing code` throws an exception amounts to setting up a **`catch`** block for an exception of the expected type. Pattern try/catch's main strength is its *flexibility*: it can check any features of any number of thrown exceptions, at any point during the execution of the test, and it can even check that a certain exception is *not* thrown (by failing inside a **`catch`** block). On the flip side, the required boilerplate code may result in tests that are *verbose*.

2) Pattern test: Version 4 of JUnit (released in 2006)[c] was the first providing a custom feature to write exceptional tests: by adding a parameter `expected` to the `@Test` annotation that marks JUnit tests, programmers can specify which tests are expected to throw which exceptions. Pattern test can make exceptional tests very *concise* and *readable*. However, it has limited flexibility: it is impossible to express which *part* of the `testing code` is expected to throw an exception, to test for *multiple* exception types, or to specify any *attributes* of the thrown exceptions other than their type.

3) Pattern rule: Version 4.7 of JUnit (released in 2009) introduced a new way of writing exceptional tests, using a field of type `ExpectedException` marked with annotation `@Rule`. Any test can set up such field to declare that the `testing code` is expected to throw an exception of a certain type. Pattern rule is somewhat more flexible than pattern test, since we can specify *attributes* of the expected exception other than its type (for example, its *message*). It can also designate that a *specific* statement of the `testing code` should throw an exception: this is the statement immediately following the calls to methods of class `ExpectedException`. However, patterns test and rule share the limitations that all code in `testing code` following the statement that throws the first exception will be ignored, and that they cannot specify *multiple* exception types in the same testing method. Tests written according to pattern rule remain *concise* but are stylistically quite different from JUnit's run-of-the-mill idioms that usually assert the expected outcome *after* the testing code rather than before it. This may be the reason why this pattern was removed from JUnit as soon as `assertThrows` became available.

4) Pattern assert throws: Static assertion method `assertThrows` was first introduced in JUnit 5.0 (released in 2017) and then added to JUnit 4.13 in 2020. Since `assertThrows` inputs the `testing code` as a *lambda*, this pattern is expressible only since Java 8. Pattern assert finally combines *conciseness* and *flexibility*, since `assertThrows` also returns the thrown exception object, which can be further inspected in the test code. It also blends with the other *assertion* methods available in JUnit, and with the test idioms they support. Pattern assert can be considered the *recommended* style to write exceptional tests since JUnit 5.0 (which no longer supports[d] patterns test and rule).

5) Pattern generic assertion: Assertion libraries—such as Hamcrest, AssertJ, and Truth—provide flexible APIs to express all sorts of expected behavior—including exceptional behavior. Only AssertJ (used in Fig. 1e) among these libraries includes methods such as `assertThatThrownBy` that implicitly catch any exceptions thrown by `testing code`; the other libraries offer methods to specify properties of exceptional objects but still rely on Java's **`catch`** blocks or JUnit's `assertThrows` method to perform the actual catching. As its name suggests, pattern generic is the most *flexible* approach to writing exceptional tests. It is easy to *combine* assertion methods in chains of method calls using the so-called "fluent" style, making it easier and more *readable* to write complex tests [13]. This structure also helps *readability*, supports powerful auto-completion *suggestions* when used within an IDE, and automatically generates informative *error messages*

[c] We report release dates of stable releases.

[d] Backward compatibility is still possible through JUnit's module `Vintage`.

```
@Test
void tryCatch()
   throws Exception {
 Exception caught = null;
 try { /* testing code */ }
 catch (Exception e)
  { caught = e; }
 if (caught == null)
  fail();     // fail
 else pass(); // pass
}
```

(a) Pattern try/catch.

```
@Test(expected =
    Exception.class)
void test()
  throws Exception {
 /* testing code */
}
```

(b) Pattern expect test.

```
@Rule
ExpectedException ex;

@Test
void rule()
   throws Exception {
 ex =
   ExpectedException.none();
 ex.expect(Exception.class);
 ex.expectMessage("error");
 /* testing code */
}
```

(c) Pattern expect rule.

```
@Test void assert()
   throws Exception {
 Exception ex =
  Assertions.assertThrows(
   Exception.class,
   ()->/* testing code */
  );
  assertEquals("error",
     ex.getMessage());
}
```

(d) Pattern assert throws.

```
@Test void generic()
   throws Exception {
  assertThatThrownBy(
    ()->/* testing code */
  ).isInstanceOf(
      Exception.class)
   .hasMessage("error");
}
```

(e) Pattern generic assertion.

Fig. 1: The main coding patterns that programmers can use to test for exceptional behavior in Java.

whenever an assertion fails. Besides the dependency on an additional library, the main *disadvantages* of using assertion libraries may come from their great flexibility: when the same behavior constraint can be expressed in several different ways, it is harder to enforce a *consistent* style within a project, and more *refactoring* and debugging effort may be needed.

III. Related Work

Testing: Testing is a key technique to develop high-quality software [1], [2]. Accordingly, there has been a massive amount of software engineering research and practice dealing with all sorts of aspects related to software testing, which we can only briefly summarize here.[e] Among the empirical contributions, researchers have studied the testing practices of developers [14], the characteristics of the tests they write [15], and how they relate to the bugs that are commonly found [12]. Among the technological contributions, a lot of effort has been devoted to bringing more *automation* to testing. Framework such as the popular JUnit[7] and TestNG[8] automate test-case *execution* by providing syntactic means of defining test inputs and the expected outputs. Other tools can automate the *generation* of test inputs [16], [17] as well as of oracles—for example in the form of assertions [18]–[20].

Exceptions and exception handling: First introduced in PL/I [8], [21], *exceptions* are nowadays available in pretty much any mainstream programming language. Since exceptional behavior is often implicated with bugs, studying how exceptions are used in programs can improve our understanding of bugs and other deficiencies, and suggest ways of improving software. We now summarize the most relevant related work on exception handling published in the last few years. For more references and details, see the proceedings of the Workshop on Exception Handling [22] and the survey [7].

A lot of empirical research has investigated how exception handling is done in practice both by mining code bases [23]–[29] and by looking at programmers' habits and guidelines [30], [31]. This line of research has revealed that exception-handling code is often complex [32] and among the most poorly understood and scarcely documented parts of a system [33]. Then, it is not surprising that exceptions are commonly implicated with bugs in a variety of software including Java libraries [34]–[37], Android apps [38], [39], and cloud systems [40]. On the positive side, empirical studies of exception-related bugs can inform the development of techniques to proactively help developers finding a relevant StackOverflow post [41], or to write exception-handling code that likely follows the best practices [42].

APIs typically throw exceptions to signal incorrect calls (for instance, invalid parameters). Therefore, API misuses can often be linked to exceptions [35] or to missing exception-handling code [36]. Conversely, it is possible to *infer* preconditions by automatically mining API calls for common exceptions that signal common errors such as null-pointer dereferencing and index out-of-bounds [34], or by manually combining the messages of exception objects with other sources of API documentation [37]. The Android mobile-programming framework defines and uses numerous framework-specific exception classes. Due to the nature of mobile apps (which are event-driven and use several external resources), writing proper exception handling code in Android is not easy [38], [39], which is why test amplification is a promising technique [43] to *validate* such exception handling code.

Exceptional testing: Even though plenty of research studied testing or exceptions, only little looked at the intersection of the two topics beyond test-case generation [3], [44], [45]; as Ebert et al. [46] put it: "perhaps surprisingly, there is comparatively little work studying how exception handling code is tested or debugged in practice, with few exceptions [43], [47]". We mentioned [43] above; [47] introduces a *coverage analysis* for exceptional-handling code. The lone work among the few related to exceptional testing that is most closely related to ours is by Dalton et al. [10], which also analyzes tests and exceptional tests in 417 Java projects. Part of their analysis is based on a survey of 66 developers about their perception of exceptional-behavior testing, which is a methodology complementary to ours. Their analysis's quantitative part includes some measures similar to ours, which can be used to corroborate our findings—most notably, they found that 61% of projects with tests also include some exceptional tests, which is very close to our 66% (see Tab. II). Finally, notice that [10]'s study targets a smaller collection of Java projects, and does not analyze all categories of exceptions or the exceptional testing patterns—which are instead a key component of our results (see Sec. IV).

[e]For brevity we focus on work about the Java programming language, although research also targets testing in other programming languages.

IV. Study Design

A. Research Questions

Our overall goal is understanding how Java developers *test for exceptional behavior*. To this end, we consider three main research questions:

RQ1: How often is exceptional behavior tested?

The first research question investigates how usual exceptional testing is in Java projects—both in absolute terms and relative to testing in general.

RQ2: What kind of exceptional behavior is tested?

The second research question looks for trends in the kinds of exception classes that feature in exceptional tests, and how these affect other characteristics of the tests such as their size.

RQ3: What coding patterns are used for exceptional testing?

The third research question analyzes how exceptional tests are written, and how they use the features of the available testing frameworks.

B. Project Selection

We started from [48]'s list of 2 672 Java projects—those with the most stars hosted on GitHub between November and December 2019. GitHub stars indicate a project's popularity and are commonly used to select source code of consistent quality [39], [41], [42], [48], [49]. Given our focus on tests, we further discarded projects that 1) have no detectable JUnit tests[f] (1 128 projects), or 2) have only the two tests that Android Studio IDE generates automatically by default[9] (374 projects). This leaves 1 157 Java projects with some non-trivial tests, which are the focus of our analysis.[g] The final project selection includes large frameworks and applications, as well as projects in different domains. The filtering criteria indicate these are projects of good quality whose developers have devoted at least some effort to writing JUnit tests (a median of 91 commits of test code per project).

C. Analysis Process

We built *JUnitScrambler*: a tool to extract data about exceptional tests in Java projects. The tool works in three steps:

build	the project from its sources
discover	the test code in the project
analyze	the discovered tests for exceptional testing patterns

Step *build* looks for recipes for Java's most popular build systems—Maven, Gradle, and Ant—and uses them to compile the project and its tests. It also tries to detect the required Java version and all external dependencies. Automatically looking for this information in build recipes may fail, or a project may not use a build script that we recognize.

If a project is built successfully, step *discover* uses JUnit 5's test discovery API[11]—which can process JUnit tests in any versions—to find tests. JUnit's test discovery may work even when the build was incomplete; in addition, our tool looks for references to executed testing classes among the output of the build process. As a last resort, our tool scans every Java source file in the project, and marks as "possible tests" those that import testing libraries [50]. The combination of these three ways of looking for tests allows step "discover" to detect tests from all JUnit versions, including tests that may not be trivially found (e.g., located in directories other than a build system's default or generated by the build process).

Step *analyze* parses all discovered tests with JavaParser,[12] and processes the resulting AST and typing information to measure the characteristics of exceptional tests that we mention in Sec. IV-C. JavaParser's rich information—augmented with the dependencies collected by the build step— supports a fine-grained analysis of testing patterns and exception types. Among other things, we distinguish between usages of `assertThat` from various testing libraries (JUnit, Hamcrest, AssertJ, and Truth), can follow nested calls in test methods, and can often determine whether exception classes from external libraries are checked or unchecked.

Measured data: For every *project*, *JUnitScrambler* reports its build system, JUnit version, whether any tests were found, and the list of classes with testing code. For every test (that is, testing method), it measures its size in non-blank lines of code (LOC), and if it detected any exceptional testing coding patterns (Sec. II-B). For every exceptional test (that is, when a pattern was found), it reports the detailed structure of the pattern, whether the test asserts on an exception message or cause, and the fully-qualified exceptional static types of the exceptions mentioned in the test (which determine their *origin* and *kind*, see Sec. II-A), and other contextual information such as any messages in the assertions or code comments.

JUnitScrambler records the raw measured data in CSV format. We then imported the data into R[13] and used it to perform the statistical analysis reported in Sec. V.

Qualitative analysis: A "Closer Look" section complements the quantitative analysis with qualitative findings about each research question, which we obtained by systematically inspecting the top-10 projects with the "most conspicuous" characteristics relevant to the question. For example, RQ1's closer look inspects projects with "the largest number of exceptional tests" and with "few exceptional tests".

V. Results

Before we delve into the details of exceptional testing, let's overview some overall characteristics of the projects we considered. Tab. I summarizes the main data.

		BUILD SYSTEM			
	ALL	MAVEN	GRADLE	ANT	NA
#	1 157	521	464	39	172
%	100.0	45.0	40.1	3.4	14.9

TABLE I: The number # and percentage % of ALL analyzed projects that use each BUILD SYSTEM. Percentages do not add up to 100% because a few projects use multiple build systems.

[f]The restriction to JUnit is justified by its popularity: we found a mere 6 projects out of 2 672 with tests written exclusively for the TestNG framework.

[g]We also ascertained that the patterns identified in Sec. II-B cover most of the exceptional tests: for example, we found only 5 projects with exceptional tests using library `catch-exception`[10], which is not covered by the patterns.

Overall, we analyzed 1 157 projects with tests. Most projects use one of three build systems: Maven is the most widely used (45.0% of projects), followed by Gradle (40.1% of projects), whereas only 3.4% of projects use Ant, and 14.9% of projects use no build system that we could detect (NA in Tab. I).

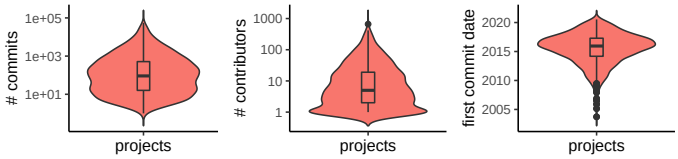


Fig. 2: Violin plots of the analyzed projects' total number of commits, number of contributors, and initial commit date of their test code. Vertical scale is logarithmic in first two plots.

Fig. 2 displays other overall characteristics of the test code among the 1 157 projects that we analyzed. The total number of *commits* varies widely among projects: its median is 91, its mean is 1 094, and its maximum 55 090 commits. The number of *contributors* also varies widely: its median is 5, its mean is 22, and its maximum is 663 contributors. The *age* of the test code, measured as the date of test code's first commit, is less spread out: its median is 2015-12-12, close to its mean 2015-07-08; nonetheless there are several outlier older projects: the oldest commit date is more than 17 years ago (2003-09-26).

A. RQ1: How often is exceptional behavior tested?

RQ1 asks how much exceptional testing is usually carried out in Java projects. As shown in Tab. II, 66.2% of the projects with *some* tests also include *exceptional* tests. The split between exceptional and regular tests is, however, not even: only 13.2% of all tests are exceptional—making up 14.6% of all lines of testing code. On the other hand, there is a strong positive correlation (Kendall's $\tau = 0.7$) between number of tests and number of exceptional tests that each project includes, which indicates that exceptional tests are an integral part of the test-writing effort in the analyzed projects.

	#PROJECTS	#METHODS	#CLASSES	$\sum$ LOC	$\overline{\text{LOC}}$
ALL TESTS	1 157	1 123 846	171 011	14 023 852	13
EXCEPTIONAL TESTS	766	148 063	41 537	2 046 930	14
% EXCEPTIONAL/ALL	66.2	13.2	24.3	14.6	110.3

TABLE II: Number of PROJECTS with some tests, number of testing METHODS and CLASSES, total $\sum$ LOC and per-method mean $\overline{\text{LOC}}$ size of test methods in lines of code. The first row comprises ALL TESTS, the second only EXCEPTIONAL TESTS, and the third the latter as a percentage of the former.

The violin plots in Fig. 3a provide more information about the effort that is usually devoted to writing exceptional tests. By comparing the two shapes in the leftmost plot, we notice that the distribution of number of exceptional tests is wider around the median. Thus, there is less inter-project variability in the number of exceptional tests compared to all tests. A similar trend exists for the total size (in lines of code) of all tests compared to exceptional tests—even though the mean size of any exceptional test is 110.3% that of any test. Indeed, there is a small but definite positive correlation (Kendall's $\tau = 0.1$) between a project's number of tests and their median size, but a negligible correlation ($\tau = -0.02$) between a project's number of exceptional tests and their median size. Thus, exceptional tests tend to be more homogeneous in size across projects, indicating that writing exceptional tests is an activity that receives significant effort but is somewhat more "standardized" than writing tests in general.

Two thirds of the projects with tests also include exceptional tests; the latter vary less in number and size.

RQ1: A Closer Look at Some Projects

We observed that projects with the largest number of tests typically also have the largest number of exceptional tests. In particular, the project with the most tests (*Eclipse Collections*[14] with over 47 283 tests) is also the project with the most exceptional tests (7 839 tests). To produce such a huge number of tests the project uses *code generation*[15] which can automatically produce variants of tests for classes that have a similar behavior (e.g., they implement the same interface).

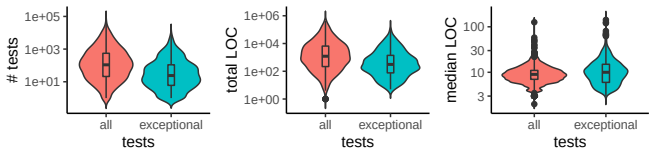
Effective large-scale testing (including exceptional testing) requires clear guidelines and effective practices. Project *Apache Geode*, for example, comprises tests in 5 different categories[16] (including unit, integration, and acceptance) and explicitly recommends how to catch exceptions in unit tests;[17] this might explain why it ranks 4th and 5th among our projects with the largest number of tests and exceptional tests. More generally, projects with the largest number of (exceptional) tests usually recommend providing unit tests when opening an issue or contributing code, and actively try to include tests with high code coverage (for example, project *Hazelcast*'s[18] tests cover over 85% of all project code according to SonarCloud;[19] the project ranks 3rd and 8th among our projects with the largest number of tests and exceptional tests).

At the opposite end of the spectrum, projects with few (exceptional) tests tend to be younger, less established, and provide simpler, more limited functionality. Project *Rest Countries*,[20] for instance, offers a REST API exporting data about worldwide countries (e.g., their currency) to add internationalization support to web applications. Tutorials and extensive examples are another group of projects with a limited number of tests and exceptional tests. It is reasonable to expect that those projects that will undergo further development will also considerably extend their test suites as they mature; citing project *Processing*'s documentation: "someday" they will have "hundreds of unit tests [...] but not today".[21]

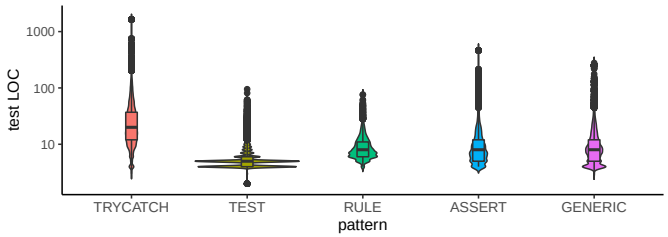
B. RQ2: What kind of exceptional behavior is tested?

RQ2 asks what kind of exceptional behavior is most frequently tested in Java projects. The characteristics of the *exception classes* in exceptional tests are a proxy for such behavior.

Categorization of exceptions: We classified the exception classes that we found in our projects according to their *origin*, *kind*, and *usage* (see Sec. II). The classifications into origin and kind are objective and thus automatic. In contrast, an exception class's intended usage is described in the class's



(a) Analyzed projects' total number of tests, total size of all tests, and median size of a test in lines of code. Each plot shows data about *all* tests next to data about *exceptional* tests.



(b) Violin plot of the analyzed exceptional tests' size in LOC grouped by the patterns they use.

Fig. 3: Violin plots of: (3a) the projects' size measures; (3b) patterns used in their tests. Vertical scales are logarithmic.

documentation and other artifacts where it features; therefore, it is somewhat informal and potentially subjective. To manage this threat, we proceeded as is customary in studies of Java exceptions classes [31], [34], [41], [44] and manually classified the usage of *only standard* Java exceptions—precisely, the same list of Hassan et al. [31]. Furthermore, we only considered categories failure and fault, since usage category return is most of the times sporadic and context-dependent—rather than being an exception class's intrinsic characteristic.

	ORIGIN			KIND		USAGE	
	Java	external	local	checked	unchecked	failure	fault
% EXCEPTIONS	6	12	82	40	60	74	25
% TESTS	76	2	26	36	70	47	57
% PROJECTS	95	32	59	81	90	89	83

TABLE III: Each column lists the percentage of EXCEPTIONS, TESTS, and PROJECTS that feature exception classes with certain characteristics: defined in Java's standard libraries, in a different external library, or locally to the project; checked or unchecked; used to signal failure or fault.

The top data row in Tab. III summarizes the result of our classification. Out of 5 152 exception classes found in our tests: 1) 82% are project-local; 6% are Java standard exceptions, and the remaining 12% are from external dependencies; 2) 40% are checked types, and the remaining 60% are unchecked. Finally, according to our classification of their intended usage, 74% of all *Java* exceptions are for failures and 25% are for faults.[h]

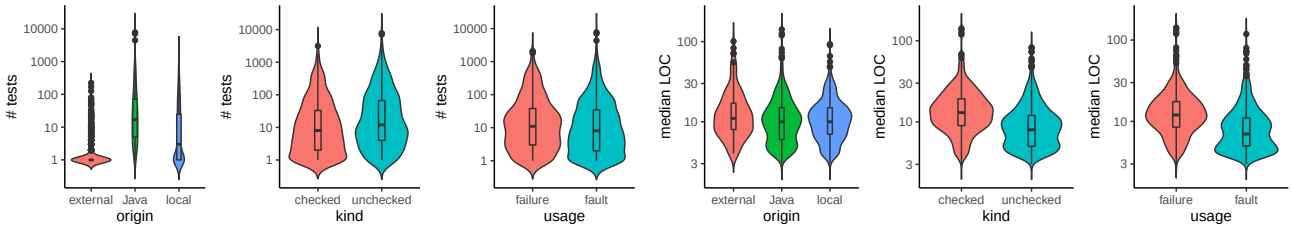
Origin: Even though Java standard exceptions are only 6% of all tested exception classes, they are by far the most widely used: 76% of all exceptional tests target an exception of *origin* Java, and 95% of all projects include at least one such test.[i] This indicates that the familiar Java exception classes are tested extensively. It may also suggest that the bulk of the exceptional behavior of most projects is not very project-specific—because the project defines no exception classes or tests them indiscriminately using abstract Java exception types. Still, 59% of all projects also test for locally defined exceptions; and about 32% of all projects also test for exceptions from external libraries. However, external exceptions feature in only 2% of the tests—and, indeed, the distribution of number of tests for each origin per project in Fig. 4a says that most projects have no more than 10 tests targeting external exceptions. This suggests that the exceptional behavior of external libraries is seldom tested specifically—possibly because libraries mainly expose standard exceptions, or developers prefer testing abstract Java exception types when dealing with third-party code.

Kind: According to Java's official documentation,[22] checked exceptions should be used when the "client can reasonably be expected to recover from [the] exception". This guideline is somewhat informal, and as a result the role of checked vs. unchecked exceptions has long been a controversial point [39]. We found that projects test for unchecked exceptions (90% of all projects, and 70% of all tests) more frequently than for checked exceptions, but the latter still feature prominently—and nearly 72% of all projects include tests involving *both* checked and unchecked exceptions (not shown in Tab. III). The minority of projects that only test for checked (10%) or unchecked (19%) do an overall limited amount of exceptional testing targeting at most a dozen exception classes. Since the compiler checks that programs include handling code for checked exceptions, *less* testing might be needed for checked exceptions thanks to these static checks. Nonetheless, we found no clear support for this expectation: distributions of the number of tests per project are qualitatively similar for checked and unchecked exceptions (Fig. 4a); and exceptional tests targeting checked exceptions are usually considerably *larger* than those targeting unchecked exceptions (Fig. 4b). In all, the checked vs. unchecked debate is far from being settled, and in practice it seems programmers use any kind of exception classes without rigid rules.

Usage: Even though we classified 3 out of 4 Java exception classes as "usage failure", projects test for both kinds of exceptional behavior—failure and fault—about as frequently, as confirmed by the qualitatively similar distributions in Fig. 4a. However, the median method testing for exceptions signaling failure is nearly twice as big as one testing for fault (16 vs. 9 LOC). A fault indicates a bug in the program—which should never occur—whereas a failure is often due to a transient condition that may be recovered from. Therefore, a test that finds a fault might not have much to do besides signaling it to the programmer for debugging, and hence it is shorter than

[h]The missing 1% is a rounding error due to class `java.io.EOFException` that we consider of usage return and we discuss in Sec. II and below.

[i]These percentages don't add up to 100% because a test or project may target multiple exceptions of different origins.



(a) Number of tests targeting various exception classes

(b) Median size in LOC of tests targeting various exception classes

Fig. 4: Violin plots of the analyzed projects' total number of tests and median size of a test. Each plot groups data according to various characteristics of the exceptions featured in tests: their *origin* (Java's standard libraries, external, or local to the project), their *kind* (checked or unchecked), and their *usage* (signaling failure or fault). Vertical scales are logarithmic.

a test that finds a failure and may try to see if the same calls in different program states lead to different behavior.

Java exception `java.io.EOFException` is mainly used to "signal end of stream"; however, "many other input operations return a special value on end of stream rather than throwing an exception",[23] which makes `EOFException` a class used primarily to pass an additional `return` value rather than to signal truly exceptional behavior. We inspected the tests targeting this exception in projects *Apache Hadoop*[24] and *ExoPlayer*[25] (two large projects among those that test for this exception) and confirmed that `EOFException` can be thrown as part of a program's normal operation: the tests of both[26,27] include an *empty* catch block for `EOFException` including a comment that the exception is "expected".

Java standard exception classes are the most frequently tested; unchecked exceptions are tested more frequently than checked ones; exceptions signaling failure and fault are tested about as frequently.

RQ2: A Closer Look at Some Projects

Kind: Larger projects with many tests invariably target exceptions of both kinds. Project *Spring Framework*[28]—a widely-used Java framework—is an interesting example because it is designed so that it only throws unchecked exceptions.[29] Nonetheless, 18% of its 2 460 exceptional tests target 34 checked exception classes—including several local to the project. This confirms that it is practically impossible to stick to only one kind of exceptions, since the roles of checked and unchecked exceptions are irredeemably intermingled in Java.

Errors (subtypes of `java.lang.Error`) are a distinct category of unchecked exceptions reserved for "serious problems that a reasonable application should not try to catch"[30] such as `OutOfMemoryError`. Projects that primarily test for errors frequently deal with low-level features of system programming, such as virtual machines (*Oracle GraalVM*[31] and *Eclipse OpenJ9*[32]), core language features (*Apache Commons Lang*,[33] *Google Guava*,[34] and *Apache Flink*[35]) and language manipulation and translation (*Google J2ObjC*[36]). Dealing with low-level features, these projects' tests handle errors to check robustness in different conditions of the runtime environment they execute in. A clear example is a test in *Apache Flink*[37] that tests the behavior of integer overflows but also includes an empty catch block for `OutOfMemoryError` with the comment "this may indeed happen in small test setups. We tolerate this".

Specific vs. generic exception: Exception-handling code should be exception-class specific,[38] whereas **`catch`** block with exception types high up in the inheritance hierarchy (such as `Exception` or even `Throwable`) are considered an anti-pattern. Since abstract types `Exception`, `Throwable`, and `RuntimeException` are among those featuring most frequently in exceptional tests (respectively, 2nd, 6th, and 10th), this anti-pattern may also occur in testing code. By manual analysis of a few larger projects, we found at least a couple of instances. Ten out of 12 of project *Saturn*'s[39] test classes deal exclusively with type `Exception`; on closer inspection, these are *integration* tests, which need not differentiate between specific exception types (a task for unit tests) but just detect thrown exceptions [51]. Another instance is project *Apache Flink*, which includes a very long method[40] with 150 try/catch blocks all for type `Exception`. In this idiom, a specific exception type `Specific` is checked with an assertion that `e` **`instanceof`** `Specific` in the **`catch`** block.

Static analysis and tests: a key usage of exceptions is signaling runtime faults; however, some faults can also be detected statically by source-code analysis. Take a relatively basic but widespread bug: accessing a **`null`** reference. Java uses exception `NullPointerException` to signal "attempts to use **`null`** in a case where an object is required";[41] several static analyzers, such as *Infer* [52] and *lgtm*,[42] automatically detect such faults. We looked at 10 of the projects with the most tests that also use *lgtm* to detect and fix errors in their code base. All of them still include exceptional tests targeting `NullPointerException`, but the tests sometimes cover corner cases that are hard to catch using static analysis, or where using **`null`** is acceptable or even expected [39]. Project *Apache ActiveMQ*'s[43] iterators, for example, throw a `NullPointerException` in some conditions when the iterator is no longer valid; in this scenario,[44] the exception doesn't signal a fault but rather returns information to the caller. Some parts[45] of *Apache Geode* also use `NullPointerException` for inter-method communication. A couple of exceptional tests[46] in *Hibernate ORM*[47] deal with using **`null`** to initialize object structures with circular references—which is notoriously tricky to analyze statically [53], [54]. In *Apache Kafka*,[48] a test[49] for `NullPointerException` captures an error that occurs when

incorrectly nesting serializers—another scenario that is likely to be off-limits for common static analysis algorithms.

C. RQ3: What coding patterns are used for exceptional testing?

RQ3 analyzes the coding *patterns* (Sec. II-B) that are used in exceptional tests, and relates them to other project features.

	try/catch	test	rule	assert	generic
% TESTS	42	32	5	9	19
% PROJECTS	83	63	24	13	39

TABLE IV: Each column lists the percentage of exceptional TESTS and PROJECTS that use any of the 5 coding patterns try/catch, test, rule, assert, and generic (see Sec. II-B).

Tab. IV shows that the most widely used pattern is try/catch, which features in 83% of all projects and 42% of all tests, followed by pattern test. In contrast, patterns rule and assert are the least frequently used—by 24% and 13% of all projects and in 5% and 9% of exceptional tests. Pattern rule's atypical syntax (see Sec. II-B) may explain why it's not widely used.[j]

Multiple patterns: The percentages in every row of Tab. IV add up to more than 100%, because any one project or test may use more than one pattern. Mixing multiple patterns in the same *test* is not common: 94% of all tests stick to a single pattern; the remaining 6% typically combine patterns try/catch or assert with assertions using generic that look for more specific exception types. A handful of tests combine three patterns, but these are outliers that make up only 0.1% of all tests. In contrast, it is common that a *project* includes tests using different patterns (see Tab. V), even though only 2% of projects include some tests for *every* pattern, and a substantial number of projects use a single pattern: 19% of projects only use pattern try/catch and 11% of projects only use pattern test.

To understand which characteristics of a project are associated with using more or fewer patterns, we fitted a Poisson[k] regressive model using a project's number of different patterns as *outcome* variable, and the number of contributors, the number of tests,[l] and the testing code's age (measured by its earliest commit date) as *predictors*. The estimated slope coefficients for number of contributors and tests are positive with 95% probability; hence projects with more contributors and tests also tend to use more exceptional testing patterns.

Patterns and size: Fig. 3b pictures the distribution of size (in LOC) of exceptional tests grouped according the patterns they use. It confirms the intuition that some patterns lead to more concise code than others. Pattern test is by far the most concise: tests using it are on average 6 lines, and their distribution is wider around and below the mean. Tests with pattern try/catch, in contrast, are on average 33 lines, and their distribution spreads over a wide range of lengths with many outliers. Pattern try/catch is the most flexible, doesn't depend on any external library, and is used in combination with other patterns; a large spread in test size is thus not surprising. Patterns assert and generic are also quite flexible, which explains the outlier tests that reach large sizes; on the other hand, their "natural" usage leads to concise tests (on average, around 12–13 lines long) which make up the bulk of the distribution.

To understand which characteristics of a test are associated with its size, we fitted a negative binomial[m] regressive model using a test's LOC size as *outcome* variable, and, as *predictors*, the *patterns* it uses, the *origin* and *kind* of the exception classes it features (see Sec. II-A), and whether it includes assertions on the *message* and *cause* of any exceptions; to control for overall project size, we also included the *total size* of the project's tests as a predictor. The results (see Tab. VI) indicate that all predictors are strongly associated with a test's size. The strongest effect is that of *patterns*: tests using pattern try/catch are the largest; those using patterns rule, assert, and generic are 36%, 39%, and 40% the size; and those with pattern test are the smallest at 18% the size. The association between size and the exceptions' *origin* is clear but less prominent: tests featuring Java standard exceptions are 92% the size of those featuring external exceptions, whereas those featuring project-local exceptions are 104% the latter's size. Somewhat counterintuitively, tests inspecting exception messages or causes (columns MSG and CAUSE in Tab. VI) tend to be smaller than those not inspecting them. There is a positive association between a test suite's overall size (controlling for a confounding effect) and each exceptional test's average size but the effect is small in comparison with the others. Finally, tests targeting unchecked exceptions tend to be smaller than tests targeting checked exceptions, but the effect is small ($1\% = 100 - 99$ reduction in size). In all, the patterns capture different ways in which developers write tests trading off conciseness, expressiveness, and flexibility.

Patterns and checked/unchecked: Projects that only follow pattern try/catch often disproportionately use checked exception classes: on average, a project in this group includes 1.5 as many tests for checked exceptions than for unchecked, and 29% of these projects *only* test for checked exceptions; in contrast, projects that do *not only* use pattern try/catch include, on average, 0.4 fewer tests for checked exception than for unchecked, and just 4% of them only test for checked exceptions. Since checked exceptions must be either caught or explicitly propagated, a try/catch block is often necessary in exceptional tests targeting checked exception, which may make using other, more concise patterns redundant.

Projects that *exclusively* use pattern test in exceptional tests show the reverse tendency, namely they primarily test for unchecked exceptions: 78% of these projects *only* test for unchecked exceptions; in contrast, just 14% of projects that do *not only* use pattern test only test for unchecked exceptions. Pattern test is a natural choice to write concise exceptional tests; indeed, exceptional tests in projects using only this

[j] Even one of the developers who built this mechanism into JUnit admits that he rarely uses it.[50]

[k] Suitable for "counting" outcome variables [55].

[l] Using all tests or only exceptional tests lead to similar conclusions.

[m] A negative binomial generalizes a Poisson for outcome variables that are overdispersed [55], like test size in our case ($\mu = 24 \ll 1751 = \sigma^2$).

%	18.6	17.4	10.8	10.7	9.8	7.0	5.9	3.4	2.0	2.0	1.6	1.5	1.3	1.2	1.2	1.2	1.1	0.8	0.7	0.4	0.4	0.4	0.3	0.1	0.1	0.1	0.1
TRY/CATCH	●	●	●	○	●	●	●	●	●	●	●	●	●	○	○	○	○	●	○	●	○	○	○	●	○	●	○
TEST	○	●	●	●	●	○	●	○	○	●	●	●	○	○	●	●	○	○	○	●	●	○	●	○	○	○	●
RULE	○	○	○	○	●	○	●	○	○	●	○	○	●	●	●	○	○	●	○	●	●	○	●	●	●	●	●
ASSERT	○	○	○	○	○	○	○	●	●	●	●	●	○	○	○	○	○	○	●	●	○	●	●	●	○	●	●
GENERIC	○	○	●	○	●	●	○	●	○	●	○	●	○	○	○	●	●	●	○	○	●	●	○	○	●	●	●

TABLE V: For each combination of patterns (those marked by ● in each column), the top row reports the percentage of all projects whose exceptional tests use exclusively that combination. Combinations not shown never occurred among the projects.

PATTERN				ORIGIN		KIND	INSPECTING		
test	rule	assert	generic	JAVA	LOCAL	UNCHECKED	MSG	CAUSE	LOC
0.18	0.36	0.39	0.40	0.92	1.04	0.99	0.73	0.83	1.06

TABLE VI: Regression estimates of each characteristic's contribution to a test's size. More precisely, each number is the *exponential* of the estimated *slope coefficient* of the corresponding predictor's variable in a negative binomial regression with outcome test size. The exponential is taken to reverse the logarithmic link function, so that the shown estimates are on the outcome scale. All predictor variables except LOC (total size of test code in the project) are dummy selector variables ($\ell - 1$ variables for a factor with ℓ possible values). All estimates are significant with 0.99 probability.

pattern are, on average, half the size of those in other projects.

Exceptional tests using **`try/catch`** *blocks are the most common and longest; those using* `@Test(expected)` *are the second most common and shortest.*

RQ3: A Closer Look at Some Projects

The largest projects that *exclusively* use pattern try/catch tend to be long-standing projects whose main development took place in the past and currently undergo only standard maintenance. Project *Joda-Time*,[51] for example, was a date-and-time library often used with older Java versions, but it is no longer maintained since its functionalities were made available in Java 8's package `java.time`. The project uses JUnit 3.8.2, which requires Java's try/catch to define exceptional tests.

Among the largest projects that exclusively use pattern test, *Algorithms*[52] (a collection of standard algorithms implementations) is a clear example of tests that privilege conciseness: 89% of its exceptional tests consist of a single call in the body, and no exceptional test's body is longer than 3 lines.

Project *SonarQube*[53]—a popular static analyzer for Java—is one of the largest projects among those that extensively use pattern rule, which features in nearly 70% of its exceptional tests. We found that this pattern coexists with others—most frequently, with generic assertions using AssertJ—to the extent that the same developer may write, on the same day,[54,55] tests using both patterns: rule for simpler tests that mainly check that a certain exception is thrown; and AssertJ fluent assertions for "deeper" tests that inspect complex exception objects.

Project *Apache Beam*[56]—a framework for data-processing tasks—is among the largest projects that use *all* 5 exceptional test patterns. It is a clear example of how large projects with many contributors (*Beam* counts 390 contributors to its test code) naturally end up with a variety of different styles of exceptional testing code. *Beam*'s class `DataFlowRunnerTest`'s Git history[57] is a microcosm of this dynamic. The class includes tests using patterns try/catch, rule, assert, and generic; different contributors (among the 19 that worked on this class) introduced tests using only one of these different patterns. In other words, each developer's preferred practice coexists with the others'.

The development history of *Beam*'s `DataFlowRunnerTest` also shines light on the interplay between availability of JUnit features and how tests are written. When, in late 2014, part of this project was first written, developers added both tests using try/catch and using rule. However, those using rule didn't take full advantage of the pattern's expressive power until two years later, when developers added assertions on the exceptions' messages. The project formally switched to JUnit 4.13—supporting pattern assert—at the end of 2018; however, tests using the new pattern were added only months later, after a period during which maintainers were aware of the new pattern but also stuck to pattern rule for the time being.[58]

Pattern assert has been available for just a few years with recent versions of JUnit (see Sec. II-B). The commit history of the largest projects that primarily use this pattern clearly show when the migration of older tests to use this new pattern took place. Project *RoaringBitmap*[59] (providing compressed bitsets) is the largest project using *only* pattern assert; in a large pull request that took place in April 2020,[60] the project migrated from JUnit 4 to 5, and updated all exceptional tests to use pattern assert. Maintainers of project *Neo4j*[61] (a popular graph database) planned the migration to JUnit 5 for over two years;[62] the migration is still ongoing,[63] but already 80% of the project's 1 671 exceptional tests use pattern assert.

Assertion frameworks such as AssertJ have supported fluent assertions, including for exceptional behavior, for years before JUnit 5 made them more widely available. Several larger projects that predominantly use pattern generic for exceptional tests started to use this pattern early on and often kept using it over JUnit 5's pattern assert even after migrating to the latest JUnit major version. Project *Spring Boot Admin*,[64] for example, only uses pattern generic, and chose to rewrite pattern test with AssertJ assertions when updating the project to JUnit 5;[65] project *Spring Initializr*[66] similarly rewrote pattern rule with AssertJ features instead of JUnit 5's assert.[67]

VI. Limitations and Threats to Validity

Threats to *construct validity*—are we measuring the right things?—are limited given that we primarily measure well-defined features (size, types, and so on). The classification in

exceptions according to their *usage* (see Sec. II-A) is more delicate; to mitigate this threat, we limited it to well-known and well-documented Java standard exceptions [31], and one author reviewed the classification made by another one.

We took great care to minimize threats to *internal validity*—are we measuring things right? Our tool *JUnitScrambler* (see Sec. IV-C) implements complementary strategies to extract useful information even from projects that are hard to build automatically without a custom environment: it parses build files to find dependencies and library versions; scans source code to detect test classes when JUnit's discovery process fails; and feeds any additional information to JavaParser to boost type resolution. Still, a few limitations remain: *JUnitScrambler* does not recognize some build systems (e.g., Bazel or Make); only processes JUnit tests; may detect an incorrect version of JUnit in projects with overly complex build processes; may miss some unusually complex combinations of fluent assertions or exceptions whose type JavaParser cannot reconstruct. We manually went through hundreds of projects and found these cases are rare—but there are a few more exceptional tests in the wild that don't feature in our analysis.

Threats to *external validity*—do the findings generalize?—mainly depend on the analyzed projects. The 1 157 projects we analyzed (selected as described in Sec. IV-B) are all open-source; it's possible the exceptional tests of closed-source industrial projects have different characteristics. Nonetheless, our projects include plenty of commits of testing code, and span a wide range of size, maturity, and application domains—from widely used Java frameworks maintained by large development teams to single-author simpler mobile apps.

VII. Applications of Findings and Future Work

Tested exceptions: A large portion (76%) of all tests we analyzed target Java standard exceptions, which are also prominent in web searches [31], StackOverflow posts [41], and mobile app bug reports [38], [39]. Unchecked exceptions `IllegalArgumentException`, `NullPointerException`, and `IllegalStateException` were among the most tested; the same classes are most frequently implicated in API misuses [35] and Android app bugs [39], and are among those with often insufficient documentation [23], [24], [37]. Thus, studying even more closely the tests featuring these exceptions in combination with the code that triggers them is an interesting direction for future work.

Messages and documentation: Undocumented exceptions (which may be thrown but are not mentioned by the documentation or signature [23]) are a common reason for uncaught exception bugs [23], [24], [39]. Exceptional tests could be a source of implicit documentation in such cases. We found that 15% of exceptional tests detail the expected message (stored by exception objects); and 17% use assertions equipped with a string that describes what the assertion checks. Both are a form of documentation[68] that could be studied using natural language processing techniques [3], [37] to help debugging and to discriminate between correct and incorrect behavior [37], [40]. For example, one test[69] in *Apache Hadoop* clearly outlines which exceptions indicate which behavior (e.g., "setting empty name should fail"), and hence it would be a valid supplement to the tested method's documentation.[70] Such tests could also identify patterns of good testing practices and serve as a guide to writing better, more thorough exceptional tests.

Wrapping: Exception wrapping—a form of propagation—is when one exception is caught and wrapped by another exception [39]. Wrapping should not be used to hide errors (which should not be handled, see Sec. V-B) within regular unchecked exceptions—which has been found may lead to crashes in Android apps [39]. We found 3% of all exceptional tests (in 19% of all projects) test on the *cause* of an exception, which indicates wrapping occurred. Such tests, although not common, may contain precious implicit information useful to debug an application's most complex exceptional behavior.

Exceptional testing patterns: The frequent usage of pattern try/catch has positive and negative implications. On the one hand, it is the only pattern that can be used with any JUnit and Java versions—including for complex scenarios. On the other hand, using try/catch with JUnit 4 is considered a code smell [11], [56]. In fact, incorrect tests using try/catch introduced silent bugs in the test suites of projects using JUnit 4 [12]. It remains that, up to JUnit 5, the expressiveness of alternative patterns test and rule remained limited (e.g., test cannot assert on messages/causes, and rule's unusual syntax is somewhat controversial [12]). In all, it is not surprising that developers of popular open-source projects have been found [11] to often prefer good old try/catch over other patterns.

Introducing empty **`catch`** blocks is considered an anti-pattern, since it is associated with more defects [28], [29], [46], [49]. Nevertheless, our data suggests that it may be legitimate in testing code (as opposed to application code): empty **`catch`** blocks featured in about 50% of all exceptional tests using pattern try/catch; more than 50% of such blocks were accompanied by an informative comment (typically: `// expected`) which explains that idiom's purpose.

Despite its limited expressiveness, pattern test remains popular with developers—probably thanks to its conciseness—to the extent that programmers introduced a "vintage `@Test`"[71] JUnit 5 extension to write pattern test even with the latest JUnit versions (which no longer support it in the base library). In contrast, more flexible JUnit features (e.g., pattern assert) have failed to reach widespread adoption (e.g., $2/3$ of projects that use JUnit versions supporting assert do not actually use the pattern). Using our dataset and our tool, we could perform a longitudinal study of how projects migrated tests from JUnit 4 to newer versions—extending the qualitative analysis at the end of Sec. V-C—to shed light on the interplay behind available features and their adoption.

Finally, studying how AssertJ is used in combination with JUnit could deepen our understanding of testing practices. We found that 39% of all projects include exceptional tests using pattern generic—usually through libraries such as AssertJ. Indeed, AssertJ is growing in popularity [13], [48] since it provides expressive and concise idioms for testing all sorts of behaviour—including exceptions [11].

REFERENCES

[1] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques: Process, Principles, and Techniques*. Wiley, 2007.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge University Press, 2007.

[3] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *ISSTA*. ACM, 2016.

[4] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE TSE*, 2019.

[5] J. Bloch, *Effective Java*, 3rd ed. Pearson Education Inc., 2018.

[6] S. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.

[7] B.-M. Chang and K. Choi, "A review on exception analysis," *Elsevier IST*, 2016.

[8] C. Ghezzi and M. Jazayeri, *Programming language concepts*, 3rd ed. John Wiley & Sons, 1998.

[9] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.

[10] F. Dalton, M. Ribeiro, G. Pinto, L. Fernandes, R. Gheyi, and B. Fonseca, "Is exceptional behavior testing an exception? an empirical assessment using Java automated tests," *EASE. ACM*, 2020.

[11] E. Soares, M. Ribeiro, G. Amaral, R. Gheyi, L. Fernandes, A. Garcia, B. Fonseca, and A. Santos, "Refactoring test smells: A perspective from open-source developers," in *SAST*. ACM, 2020.

[12] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," *IEEE ICSME*, 2015.

[13] A. Zerouali and T. Mens, "Analyzing the evolution of testing library usage in open source java projects," in *IEEE SANER*, 2017.

[14] A. Deursen, L. M. F. Moonen, A. Bergh, and G. Kok, *Refactoring test code*. CWI (Centre for Mathematics and Computer Science), 2001.

[15] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," *IEEE/ACM ASE*, 2016.

[16] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," *ACM OOPSLA*, 2007.

[17] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," *SIGSOFT/FSE 2011*, Sep 2011.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *ICSE*, 1999.

[19] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *ICSE*. ACM, 2011.

[20] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *ICSE*. ACM, 2020.

[21] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Commun. ACM*, no. 12, pp. 683–696, 1975.

[22] *Proceedings of the 5th International Workshop on Exception Handling, WEH 2012, Zurich, Switzerland, June 9, 2012*. IEEE, 2012.

[23] M. Kechagia and D. Spinellis, "Undocumented and unchecked: exceptions that spell trouble," *MSR*, 2014.

[24] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of Java libraries: an empirical study," *MSR*, 2016.

[25] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in Java?" *IEEE/ACM MSR*, 2016.

[26] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," *IEEE/ACM MSR*, 2016.

[27] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: An empirical study," *IEEE/ACM MSR*, 2016.

[28] G. B. De Pádua and W. Shang, "Studying the prevalence of exception handling anti-patterns," *IEEE/ACM ICPC*, 2017.

[29] B. Jakobus, E. A. Barbosa, A. Garcia, and C. J. P. de Lucena, "Contrasting exception handling code across languages: An experience report involving 50 open source projects," *IEEE ISSRE*, 2015.

[30] H. Melo, R. Coelho, and C. Treude, "Unveiling exception handling guidelines adopted by Java developers," in *SANER*. IEEE, 2019.

[31] F. Hassan, C. Bansal, N. Nagappan, T. Zimmermann, and A. H. Awadallah, "An empirical study of software exceptions in the field using search logs," in *ESEM*. ACM, 2020.

[32] M. P. Robillard and G. C. Murphy, "Designing robust Java programs with exceptions," *SIGSOFT/FSE*, 2000.

[33] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in Java programs," *Elsevier JSS*, 2015.

[34] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus," in *SIGSOFT/FSE*. ACM, 2014.

[35] M. Wen, Y. Liu, R. Wu, X. Xie, S. Cheung, and Z. Su, "Exposing library API misuses via mutation analysis," in *ICSE*. IEEE/ACM, 2019.

[36] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow," in *ICSE*. ACM, 2018.

[37] H. Zhong, N. Meng, Z. Li, and L. Jia, "An empirical study on api parameter rules," in *ICSE*. ACM, 2020.

[38] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in Android apps," in *ICSE*. ACM, 2018.

[39] R. Coelho, L. Almeida, G. Gousios, A. V. Deursen, and C. Treude, "Exception handling bug hazards in android," *Springer EMSE*, 2017.

[40] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding exception-related bugs in large-scale cloud systems," *ASE*, 2019.

[41] S. Mahajan, N. Abolhassani, and M. R. Prasad, "Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching," in *ESEC/FSE*. ACM, 2020.

[42] T. Nguyen, P. Vu, and T. Nguyen, "Code recommendation for exception handling," in *ESEC/FSE*. ACM, 2020.

[43] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," *ICSE*, 2012.

[44] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, "Effective and efficient api misuse detection via exception propagation and search-based testing," in *ACM ISSTA*. ACM, 2019.

[45] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Botsing, a search-based crash reproduction framework for Java," in *ASE*. IEEE/ACM, 2020.

[46] F. Ebert, F. Castor, and A. Serebrenik, "A reflection on *An Exploratory Study on Exception Handling Bugs in Java Programs*," *SANER. IEEE*, 2020.

[47] E. S. F. Najumudheen, R. Mall, and D. Samanta, "Modeling and coverage analysis of programs with exception handling," in *ISEC*. ACM, 2019.

[48] T. Nakamaru, T. Matsunaga, T. Yamazaki, S. Akiyama, and S. Chiba, "An empirical study of method chaining in java," in *MSR*. ACM, 2020.

[49] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to Handle Exceptions," *ASE*, 2020.

[50] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE TSE*, 2019.

[51] D. Holling, A. Hofbauer, A. Pretschner, and M. Gemmar, "Profiting from unit tests for integration testing," in *IEEE ICST*, 2016.

[52] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods*. Springer, 2015.

[53] A. J. Summers and P. Müller, "Freedom before commitment: a lightweight type system for object initialisation," in *OOPSLA*. ACM, 2011.

[54] B. Meyer, "The dependent delegate dilemma," in *Engineering Theories of Software Intensive Systems*. Springer, 2005.

[55] R. McElreath, *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman & Hall, 2015.

[56] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsDetect: an open source test smells detection tool," in *ESEC/FSE*. ACM, 2020.

URL REFERENCES

1. https://stackoverflow.com/questions/156503
2. https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
3. https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html
4. https://blog.jooq.org/2013/04/28/rare-uses-of-a-controlflowexception/
5. https://github.com/junit-team/junit4/wiki/Exception-testing
6. https://assertj.github.io/doc/#assertj-core-exception-assertions
7. https://junit.org/junit5/
8. https://testng.org/doc/
9. https://developer.android.com/studio/test
10. https://code.google.com/archive/p/catch-exception/

11. https://junit.org/junit5/docs/current/user-guide/#launcher-api-discovery
12. https://javaparser.org/
13. https://cran.r-project.org/
14. https://github.com/eclipse/eclipse-collections
15. https://github.com/eclipse/eclipse-collections/blob/master/CONTRIBUTING.md
16. https://github.com/apache/geode/blob/develop/TESTING.md
17. https://cwiki.apache.org/confluence/display/GEODE/About+Unit+Testing
18. https://github.com/hazelcast/hazelcast
19. https://sonarcloud.io/dashboard?id=hz-os-master
20. https://github.com/apilayer/restcountries
21. https://github.com/processing/processing/blob/master/README.md
22. https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
23. https://docs.oracle.com/javase/8/docs/api/java/io/EOFException.html
24. https://github.com/apache/hadoop
25. https://github.com/google/ExoPlayer
26. https://github.com/apache/hadoop/blob/trunk/hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/TestFSInputChecker.java#L200
27. https://github.com/google/ExoPlayer/blob/release-v2/library/core/src/test/java/com/google/android/exoplayer2/extractor/DefaultExtractorInputTest.java#L140
28. https://github.com/spring-projects/spring-framework
29. https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch11s02.html
30. https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html
31. https://github.com/oracle/graal
32. https://github.com/eclipse/openj9
33. https://github.com/apache/commons-lang
34. https://github.com/google/guava
35. https://github.com/apache/flink
36. https://github.com/google/j2objc
37. https://github.com/apache/flink/blob/master/flink-streaming-java/src/test/java/org/apache/flink/streaming/runtime/operators/windowing/KeyMapTest.java#L101
38. https://github.com/junit-team/junit4/wiki/Exception-testing#trycatch-idiom
39. https://github.com/vipshop/Saturn
40. https://github.com/apache/flink/blob/df525b77d29ccd89649a64e5faad96c93f61ca08/flink-core/src/test/java/org/apache/flink/core/memory/MemorySegmentUndersizedTest.java#L130
41. https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html
42. https://lgtm.com/
43. https://github.com/apache/activemq
44. https://github.com/apache/activemq/blob/9abe2c6f97c92fc99c5a2ef02846f62002a671cf/activemq-unit-tests/src/test/java/org/apache/activemq/broker/region/cursors/FilePendingMessageCursorTestSupport.java#L83
45. https://github.com/apache/geode/blob/develop/geode-core/src/distributedTest/java/org/apache/geode/internal/cache/execute/OnGroupsFunctionExecutionDUnitTest.java
46. https://github.com/hibernate/hibernate-orm/commit/3489f75e1d455049cffd45694f025b97487b429f,https://lgtm.com/projects/g/hibernate/hibernate-orm/rev/1e5a8d3c434c6791b89281c4ebf04ef08181fcd7
47. https://github.com/hibernate/hibernate-orm/
48. https://github.com/apache/kafka
49. https://github.com/apache/kafka/blob/trunk/streams/src/test/java/org/apache/kafka/streams/kstream/WindowedSerdesTest.java#L73
50. https://github.com/junit-team/junit4/issues/706#issuecomment-21385116
51. https://github.com/JodaOrg/joda-time
52. https://github.com/pedrovgs/Algorithms
53. https://github.com/SonarSource/sonarqube
54. https://github.com/SonarSource/sonarqube/commit/14d6de3529b12ec0af367e551cf66ac6daae1ca7
55. https://github.com/SonarSource/sonarqube/commit/e4b519ed129dbc7b76eab00d6c48166a8993e35f
56. https://github.com/apache/beam
57. https://github.com/apache/beam/blob/master/runners/google-cloud-dataflow-java/src/test/java/org/apache/beam/runners/dataflow/DataflowRunnerTest.java
58. https://github.com/apache/beam/pull/5150#discussion_r182212260
59. https://github.com/RoaringBitmap/RoaringBitmap
60. https://github.com/RoaringBitmap/RoaringBitmap/pull/396
61. https://github.com/neo4j/neo4j
62. https://github.com/neo4j/neo4j/commit/0ce66ab6ebd454f9dbb5a0cf36e0f2483edec413
63. https://github.com/neo4j/neo4j/pull/12444#pullrequestreview-398471953
64. https://github.com/codecentric/spring-boot-admin
65. https://github.com/codecentric/spring-boot-admin/commit/caef5a004cbbc4ba897d854094b2546efd15d52b#
66. https://github.com/spring-io/initializr
67. https://github.com/spring-io/initializr/commit/2816c216315b989c45c25c18fd9f72bb606db8ee#diff-e49dd42170d49f6c1eb73139645c48cf
68. https://rules.sonarsource.com/java/RSPEC-2698
69. https://github.com/apache/hadoop/blob/2ba44a73bf2bb7ef33a2259bd19ee62ef9bb5659/hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/server/namenode/FSXAttrBaseTest.java#L274
70. https://github.com/apache/hadoop/blob/2ba44a73bf2bb7ef33a2259bd19ee62ef9bb5659/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/fs/FileSystem.java#L3049-L3062
71. https://junit-pioneer.org/docs/vintage-test/