Towards Code Improvements Suggestions from Client Exception Analysis

Diego Marcilio Carlo A. Furia
dvmarcilio.github.io bugcounting.net
Software Institute – USI Università della Svizzera italiana – Lugano, Switzerland

***Abstract*—Modern software development heavily relies on reusing third-party libraries; this makes developers more productive, but may also lead to misuses or other kinds of design issues. In this paper, we focus on the *exceptional* behavior of library methods, and propose to detect *client* code that may trigger such exceptional behavior. As we demonstrate on several examples of open-source projects, exceptional behavior in clients often naturally suggests *improvements* to the documentation, tests, runtime checks, and annotations of the clients.**

In order to *automatically detect* client calls that may trigger exceptional behavior in library methods, we show how to repurpose existing techniques to extract a method's exception precondition—the condition under which the method throws an exception. To demonstrate the feasibility of our approach, we applied it to 1,523 open-source Java projects, where it found 4,115 cases of calls to library methods that may result in an exception. We manually analyzed 100 of these cases, confirming that the approach is capable of uncovering several interesting opportunities for code improvements.

I. Introduction

Any piece of client code that calls a library method must comply with the method's *precondition*. Thus, analyzing method calls against the callees' preconditions can reveal issues with how a library is used, and possibly suggest useful improvements to the client code.

A widespread scenario occurs when a library method may throw an *exception*; for example, to signal that one of its arguments should not be `null`. If we can show that a client *never* calls the method with a `null` argument, or suitably handles the exception (for example, with a `try/catch` block), we rule out a certain category of faults. Conversely, if we find a concrete client execution where the exception is raised and not handled, this suggests a number of improvements to the client code, such as adding tests or documentation for this possible exceptional behavior, or perhaps modifying the client so that it handles the exception directly. As we discuss in Sec. II on concrete examples of Java code, such scenarios of client code calling library methods that may throw exceptions are quite common in open-source projects; analyzing them systematically (and automatically) has the potential of revealing interesting instances of misuses and critical cases, as well as of suggesting ways of *improving* the client code to address the issues.

Unfortunately, there are two main obstacles that stand in the way of practically pursuing this idea of *analyzing exception preconditions of library methods in client code to suggest code improvements to the client*. First, (exception) preconditions are often documented only informally (e.g., using natural-language comments) and partially [10], [17]. Second, even if we have a formula precisely expressing a library method's exception precondition, determining whether a call to the method may actually raise an exception requires precise reasoning about the client code (for example, through symbolic execution), which remains challenging to carry out on code bases of realistic size and complexity.

In Sec. II of this paper, we discuss a practical approach that can deal with these two difficulties. To this end, it leverages recent work on automatically *extracting* exception preconditions in a way that is scalable (applicable to realistic projects) and precise (always returns correct exception preconditions) [13], [15], [18]. Running these tools on widely used Java libraries and frameworks (including a substantial portion of the JDK) populates a database of exception precondition Boolean formulas, which precisely indicate under what conditions calling a certain library method results in an exception. To pursue our approach, we then discuss how to repurpose these existing detection techniques so that they can run on *client* code and find *feasible matches* of any exception preconditions in their database; in other words, they detect calls to any of the methods with an exception precondition that may result in an exception being thrown.

Sec. III describes preliminary experiments that we conducted to assess the practical feasibility of our approach. Among the aforementioned tools for exception precondition detection we used WIT [13] for our experiments. First, we selected 1,523 open-source Java projects that use some of the libraries that can be analyzed with WIT. Then, we modified WIT to detect feasible matches of exception preconditions—that is, possible exceptional behavior in the clients—and ran it on the selected projects. We found 4,115 such matches, which indicates that our analysis is widely applicable. We also manually analyzed a random sample of 100 matches, in order to better understand what kinds of issues the matches reveal, and how they could be turned into actionable suggestions for improvements to the client code, its tests, or its documentation. We report several concrete examples taken from open-source Java projects, which lend weight to this paper's core idea: identifying possible improvements to client code by automatically analyzing the exception preconditions of library methods.

As we discuss in Sec. IV, while there is plenty of related

Work partially supported by SNF grant 200021-182060 (Hi-Fi).

Listing 1: Documentation of `java.util.Random.nextInt(int)`.

```
// @throws IllegalArgumentException if bound is not positive
public int nextInt(int bound) {
  if (bound <= 0) throw
    new IllegalArgumentException("bound must be positive");
  // ...
}
```

Listing 2: Client code calling Lst. 1's method.

```
public static int random(final int min, final int max) {
  return Utils.RANDOM.nextInt(max - min) + min;
}
```

work about analyzing exceptional behavior and detecting API misuses, the combination of the two concepts has hardly systematically been explored. Therefore, this paper's ideas follow an original, under-researched direction, and Sec. III's experiments assess its potential.

II. From Exception Preconditions to Code Improvements

This paper's main idea is an approach to automatically analyze client code for potential throws of exceptions in library methods. Precisely, a method `m`'s *exception precondition* is a Boolean condition $P_{\mathtt{m}}$ under which the method terminates with an exception. Conversely, a *potential throw* ("*pothrow*" for short) is a piece of client code with a call to `m` whose actual arguments *may* match `m`'s exception precondition $P_{\mathtt{m}}$, and that *does not* handle the corresponding exception. Potential throws point to client code that may not fully conform to the library's (exceptional) specification—a possible case of design issues or even misuses. Sec. III describes some experiments supporting our hypothesis that pothrows in real projects can indeed suggest code improvements and refactoring. The rest of this section outlines an approach to detect pothrows automatically.

A. An Example of Potential Throw Detection

Lst. 1 shows Java's `java.util.Random.nextInt(int)`,[a] a library method that throws an `IllegalArgumentException` (IAE) if its argument is strictly less than one. Even a basic exception precondition like `nextInt`'s (explicitly documented in the method's Javadoc natural language documentation) may be non-trivial to handle properly for clients. For example, consider method `random` in project Zelix Injection;[b] as shown in Lst. 2, `random` is a pothrow of `nextInt`: if `max` $\leq$ `min`, the call to `nextInt` fails with an uncaught IAE whose error message is not very informative in the client's context.

Listing 3: Possible improvements to Lst. 2's code (in color).

```
// @throws IllegalArgumentException if @code\{max <= min\}
public static int random(final int min,
  @Refinement("max > min") final int max) {
    Validate.isTrue(max > min, "max <= min");
    return Utils.RANDOM.nextInt(max - min) + min;
}

@Test
void random_throws_IAE() {
  assertThrows(IllegalArgumentException.class,
    () -> Utils.random(1, 1);
  // ... other cases
}
```

B. Code Improvements

Even though code including potential throws might be perfectly correct, more commonly it indicates possible design issues, which, in turn, may suggest improvements to the code, its documentation, or its tests that increase its quality for its own clients and for the whole project. In fact, there is evidence that exceptional behavior is often insufficiently documented and tested even *within* a project [19], [12]; the same issues are likely to intensify when considering a project's *clients*.

Lst. 3 shows four possible improvements for Lst. 2's pothrow. *Documenting* the derived exception precondition of `random` with a `@throws` tag helps its users know when to expect an exception. *Argument checking* (using `Validate.isTrue` in Lst. 3) performs a runtime check that `random`'s actual arguments will not trigger an exception; this follows the *fail fast* principle, signaling precisely the condition and location of the exception when one occurs. Extended *type annotations* (using Liquid Java's `@Refinement` annotation in Lst. 3) go one step further as they support checking for possible exceptional behavior *at compile time* using tools such as the Checker Framework [16] and Liquid-Java [7]. Providing *tests* that exercise exceptional behavior (through JUnit's `@Test` in Lst. 3) also helps code quality, as it provides means to detect possible regressions, and serves as a concrete counterpart to the method's documentation.

C. Detecting Potential Throws Automatically

In order to automatically find instances of pothrows, we propose an approach in three steps. First, we collect exception preconditions of library methods; to this end, we can use any recently developed static techniques [13], [15], [18] that are applicable to realistic projects and return *correct* exception preconditions (if the preconditions may be incorrect, the whole analysis would become noisy and imprecise).

Second, we analyze *clients* of the libraries, looking for *calls* to any of the library methods for which exception preconditions are available. Ideally, this step would be performed *without* fully building the client code, so that the analysis is more lightweight and can also target parts of a project. Here too, we privilege precision (every match is a real match) over recall (all possible matches are detected).

Third, we determine whether the arguments of any calls identified in the previous step may actually satisfy any of the available exception preconditions. This amounts to a *feasibility* check that finds a condition over the client's arguments (such as `max` $\leq$ `min` in Lst. 2) that triggers the exception. In general, the feasibility check requires precisely reasoning about client code at its call locations. For example, if method `random` in Lst. 2 called `nextInt` with argument

`1 + Math.max(min, max) - Math.min(min, max)`, it should recognize that this expression is always positive, and hence `nextInt` will not throw. The feasibility checks should be precise as well (since we do not want to report many false alarms), but they should also achieve a reasonable recall—otherwise, the analysis would produce hardly any output. To perform the feasibility check, we encode it as a modular variant of the same exception precondition detection performed in the first step: given a piece c of client code calling library method `m`, determine c's exception precondition using `m`'s. Any such exception preconditions of c are reported as *pothrows*.

III. Experimental Evaluation

In this section, we first discuss our prototype implementation of our approach to detect potential throws of library exceptions in client code (Sec. III-A); then, we present the design (Sec. III-B) and quantitative results (Sec. III-C) of an empirical evaluation on several open-source Java projects; finally, we discuss several interesting cases that emerged in these experiments, which we manually inspected to validate the approach and to illustrate its practical usefulness (Sec. III-D).

A. Potential Throw Detector Implementation

Among the available techniques for exception precondition detection, we used WIT [13] as the basis for our implementation. When run on a library, WIT returns two kinds of exception preconditions—called *expres* and *maybes* in [13]. For our work, we only consider the former, which pass a path feasibility check, and hence are *correct* by construction.

First, we added support to store in a database the exception precondition WIT collects over multiple runs, so that they can be queried by library and method signature. Second, we wrote a simple program that uses JavaParser[c] to scan through a project and resolve the fully qualified names of any called library methods, and then searches the database of exception precondition for any match of these called methods. Third, we modified WIT so that it analyzes any enclosing method in the client that includes a call to one of the matching library methods; WIT determines whether the callee's exception can be propagated to the caller (the client) and under which conditions; in other words, it reports potential throws (pothrows) in the client. Again, we enable WIT's feasibility checks, so that it only reports pothrows that are indeed feasible.

B. Empirical Study: Design

We ran an empirical study to confirm that our approach is applicable to realistic projects, that it can identify a significant number of pothrows, and that several of these pothrows are indicative of design issues—and potential code improvements.

First, we selected 21 widely used open-source Java libraries including 6 analyzed in WIT's original work [13] (joda-time, and Apache Commons Lang, IO, Text, Configuration, and Math), as well as 15 new ones (Java 11's[1] JDK, Apache Commons Codec and Collections, Eclipse Collections, ehcache3, gson, Guava, hibernate-orm, jaxb-ri, jsoup, retrofit, and Spring boot, data-jpa, framework, and security).

We also selected several *client* projects from two different sources. Using the GHS search tool [4], we gathered 1,312 Java (non-fork) projects on GitHub with at least 10 stars and a thousand lines of code. We did not perform any a priori check that these projects use any of the 21 libraries we considered; however, it's overwhelmingly likely that these projects at least use some JDK library classes (e.g., `String`). To further increase the diversity of client projects, we also gathered another 220 client projects from the DUETS dataset [6], which consists of library/client pairs among Java open source projects developed with Maven; we specifically collected all client projects that use the latest version of joda-time, jsoup, and all Apache Commons projects we analyzed. In the following, GHS denotes the first batch of 1,312 projects, and DUETS the second batch of 220 projects.

Finally, we selected 100 pothrows among those reported in all projects and analyzed them manually. This sample of pothrows corresponds to 2.5% of all the 4,115 pothrows reported by our tool (see Sec. III-C). This is a reasonable sample size for an exploratory study, given that manual checks like this can be very time-consuming [13], [14]: they took the first author more than eight hours. We sampled opportunistically, trying to cover several different libraries, called methods, and library projects. The manual analysis was, first of all, a sanity check to confirm that the pothrows are correct (i.e., they identify method calls that *may* throw an exception). Most of the times, confirming the correctness of a pothrow was straightforward (e.g., a possible null argument), and required only a cursory analysis of the call context. For more complex cases (e.g., a call to `StringBuilder.append`[d] in project feathersui-starling-sdk [e] with arguments `empty array`, `2`, and `-1` throws an `IndexOutOFBoundsException`), we inspected the code more extensively using `jshell`.[f] After the sanity checks, we also thought about what kinds of code improvements the manually analyzed pothrows suggest; Sec. III-D presents a few selected interesting examples.

C. Empirical Study: Quantitative Results

Running WIT on the 21 selected libraries populated our database with 14,180 exception preconditions of 10,204 public library methods. The analysis of the 1,312 GHS client projects found 106,345 calls to 1,961 of the analyzed library methods. The analysis of the 220 DUETS client projects found 28,324 calls to 806 of the analyzed library methods. Overall, we found 134,579 calls matching 1,961 of the analyzed library methods (i.e., the called methods in the DUETS batch are a subset of the called methods in the GHS batch). Running our modified version of WIT on the code snippets surrounding each of these 134,579 client calls identified 4,115 pothrows (2,885 in the GHS projects and 1,260 in the DUETS projects)—around 3% of the client calls. We confirmed that all the 100 pothrows we manually analyzed were correctly identified by the tool.

We can think of a possible explanation for why only a fraction of all matching calls are pothrows. Precondition

[1] We focus on Java 11 because it's the latest Java LTS version that JavaParser fully supports.

LIBRARY METHOD	CLIENTS	CALLS	POTHROWS
`ArrayList.ArrayList(int)`	351	3 446	140
`ArrayList.get(int)`	333	8 538	11
`File.File(String)`	610	7 597	660
`Integer.parseInt(String)`	589	7 359	62
`Objects.requireNonNull(T)`	156	2 292	832
`Objects.requireNonNull(T, String)`	89	1 002	643
`Optional.of(T)`	166	1 234	36
`Random.nextInt(int)`	296	2 769	38
`String.substring(int)`	640	6 285	12
`String.String(char[], int, int)`	81	304	60

TABLE I: Ten of the most widely called library methods in our experiments. For each LIBRARY METHOD, the table reports the number of CLIENT projects with at least one call to the method, the total number of CALLS to the method, and how many of the calls are POTHROWS (potentially throwing).

inference techniques like WIT trade off recall for precision [13]; in our experiments, the modified WIT only reports a call as pothrow if it can conclusively establish that the call is feasible, which may miss some real instances. (In fact, its original evaluation [13] indicates that WIT's recall can dip below 10% on some projects.) Regardless, it is also reasonable to expect that a large fraction of library method calls are set up by the client to comply with the library's preconditions or are within a **`try`** block—and thus, they never result in an exception.

Tab. I gives an overview of ten of the most frequently called library methods among those we considered in our experiments; all of them are to JDK methods. In fact, it is clear that JDK methods dominate both the matching calls and the pothrows: overall, only 4.7% (6,371) of all calls, and 9.5% (387) of all pothrows refer to methods in libraries *other* than the JDK. Even though the DUETS projects should focus on non-JDK libraries, they still use plenty of JDK libraries: among DUETS projects, 96% (27,092) of calls, and 92% (1,164) of pothrows, refer to some of 614 JDK library methods; among GHS projects, 95% (101,206) of calls, and 90% (2,564) of pothrows, refer to some of 1,758 JDK library methods. Overall, the 6,371 calls and 387 pothrows involve only 1,007 non-JDK library methods; just three of these libraries (Apache Commons Lang, Guava, and Spring framework) account for 998 calls and 93 pothrows of 29 argument-checking library methods.

In hindsight, JDK's dominance is not surprising. First, virtually every project—even if it uses other common libraries—is a client of the JDK. Second, just because a project declares a certain library as a dependency does not mean that it uses it extensively; in fact, it may not use it at all: Harrand et al.'s empirical study [11] found that 41% of declared project dependencies do not correspond to any API usages at the bytecode-level. The study also found that, for more than half of the 94 analyzed libraries, 75% of the clients use only 12% of the libraries' methods; thus, expecting a much larger number of pothrows in our experiments is unrealistic.

D. Empirical Study: Qualitative Discussion

The constructor of `java.util.ArrayList` throws an `IllegalArgumentException` if the given initial capacity is a negative number. As shown in Tab. I, calls to this method are common in our client projects; 140 of them are pothrows, which happen when the actual argument is an expression that may be negative. (None of these pothrows is a sure bug, i.e., none of them passes a negative *literal* to `ArrayList`.) Lst. 4 shows an interesting case from project SuanShu, involving two arguments of public static method `getBasis`.[g] The client method first checks the precondition `dim` $\geq$ `nCols`, and then calls `ArrayList`'s constructor with argument `nCols`; thus, if `dim < 0`, the constructor's exception will propagate to the client. Perhaps `dim`, which should denote a dimension, is supposed to always be a nonnegative number; if this is the case, project getBasis could benefit from making this assumption explicit using a combination of the code improvements outlined in Sec. II: adding documentation, argument checking, extended type annotations, and tests to boot.

Listing 4: Pothrow call to `ArrayList`'s constructor.

```
// @throws IllegalArgumentException if there are
//         more columns requested than the dimension
public static List<Vector> getBasis(int dim, int nCols) {
  if (dim < nCols)
    throw new IllegalArgumentException(msg);
  List<Vector> basis = new ArrayList<Vector>(nCols);
  // ...
}
```

Here is another piece of evidence in support of our hypothesis that automatically analyzing potential throws can reveal subtle semantic differences between different clients and libraries. The constructor of `LinkedBlockingDeque`,[h] another `java.util` data structure, throws an exception if its initial capacity argument is negative *or zero*. Indeed, we found two pothrow calls that may violate this constraint in our analyzed projects.[i,j] Interestingly, whereas `LinkedBlockingDeque` implements interface `Deque`, other implementations of the same interface may have different exception preconditions; for example, `ArrayDeque` robustly accepts any value as initial capacity, and simply resets it to one if given a zero or negative number.[k] Thus, its clients cannot incur any pothrow when constructing instances of `ArrayDeque`.

It is well known that null pointer derefencing—signaled by `NullPointerException` (NPE) in Java—is a widespread problem in programming languages where they can happen [9]—and one that prompted countless attempts at mitigating it [5][l]. Analyzing some of the numerous instances of pothrows that may result in a NPE, we realized that the problem is compounded when **`null`** is a perfectly valid value for some arguments but not for others. Take the constructor of class `JFreeChart`[m] from the homynomous project, whose signature is shown in Lst. 5: argument `title` may be **`null`** (denoting an empty title), whereas argument `plot` results, through an indirect check, in a IAE if it is **`null`**. This instance of pothrow is already explicitly documented in `JFreeChart`'s constructor; but it still highlights the usefulness of an automated analysis that can follow third-party library dependencies and disentangle different valid usages of a method.

Listing 5: The signature and header comment of `JFreeChart`'s constructor in client project JFreeChart.

```
// @param title  the chart title ({@code null} permitted).
// @param plot  the plot ({@code null} not permitted).
public JFreeChart(String title, Plot plot) {
  // ...
}
```

Listing 6: Private method calling JDK's `Properties.setProperty`.

```
private void add(String key, String value) {
  properties.setProperty(key, value);
}
```

Besides, not all methods are as accurately documented as Lst. 5's constructor. Consider, for example, method `Properties.setProperty(String, String)`[n] in JDK's package `java.util`, which throws a NPE if any of its two arguments is **`null`**. Despite being widely used (we found 1,545 calls to it in our projects), method `setProperty`'s documentation does not mention its exception precondition. In fact, we found 12 pothrows that involve calls to `setProperty`. Lst. 6 shows one of these cases from project javapos_shtrih:[o] a wrapper of `setProperty`, which thus has the same exception precondition as pothrow. If we want to think about possible code improvements in this case, it is important to notice that `add` is *private*. While it might be called with a null argument, all its calls *within* the project supply non-null arguments; the project developers may have certain guidelines on how (and if) such cases need documentation or other kinds of annotations. In general, however, extended type checking annotations are often applied to private methods as well—for the few projects that make the effort of producing and maintaining such annotations; for example, Lst. 7 shows a snippet[p] from a project part of the popular Jenkins automation server which systematically annotates all its methods (also private ones) with `@NonNull` annotations.

The examples suggest that, once a developer identifies concrete cases of potential throws, they can apply their judgment, preferences, and project knowledge to devise the most suitable code improvements. There is usually a certain latitude in how to change the code (if it needs changing), but even seemingly minor changes may be beneficial—especially with exceptions, where "failing fast" may be the least bad option [3].

IV. Related Work

Previous work has found that (Java) exception-handling code is often undocumented [19] and tied to anti-patterns [17].

Listing 7: An example of a private method annotated with `@NonNull`.

```
private static CacheStatus getCacheStatus(
  @NonNull LibraryCachingConfiguration cachingConfiguration,
  @NonNull final FilePath versionCacheDir) {
  // ...
}
```

As APIs typically throw exceptions to signal invalid preconditions [12], uncaught exceptions are often symptoms of API misuses [2]—in turn, a common cause of bugs.

A lot of research has studied API misuses from different angles [1], often noting the relevance of exceptional behavior (which motivates our own work): for example, passing an invalid argument (e.g., null) and omitting a try-catch block are quite common [11] (but, fortunately, "there are various ways to fix bugs related to [these] API misuses"); static API-misuse detectors are often limited with respect to exceptional behavior [2]; and automated program repair approaches could benefit from better API-misuse detection capabilities [10].

Static analysis tools such as Infer,[q] Coverity Scan,[r] SpotBugs,[s] and SonarQube[t] all have rules that check for null dereferences: a null deference occurs in a piece of code like `s.length()` when `s` is **`null`**. Our approach does not report possible null dereferences in the clients as pothrows, since it is only concerned with exceptions that originate in external libraries. Conversely, to our knowledge, these static analyzers do not generally report potential throws that originate in external libraries. SonarQube does have rules that check API misuses of common Java libraries (e.g., JUnit, Spring, and Mockito), as well as the JDK core APIs; however, it lacks rules that look for general potential throws in external libraries. The only case we found that does something along the lines of our approach is SonarQube's rule "Optional value should only be accessed after calling `isPresent()`".[u] When an instance `o` of an `Optional` type is empty (i.e., `o.isPresent()` returns **`false`** or `o.isEmpty()` returns **`true`**), a call `o.get()` throws a `NoSuchElementException`.[v] The SonarQube rule looks for calls to `get()` that are not preceded by checks that `isPresent()` returns true. This is a fairly complex rule to implement, as it requires reasoning about the feasibility of individual control-flow paths; indeed, SonarQube labels it as requiring "symbolic execution".

A number of exception precondition extraction techniques have been developed in the last few years, using different sources of information—from documentation, to clients, to library code [19]—mostly using the extracted precondition to generate documentation [13], [15], [20]. In this paper, we explored the idea of using these techniques to first extract preconditions from libraries, and then to analyze client code of these libraries. To our knowledge, Zeng et al.'s recent work [18] is the only other approach that experiments with the idea of combining the results of library and client analysis. Their work is not specific to exception behavior but targets different kinds of API misuses and information sources (including Javadoc natural language documentation, call graphs, method names, and annotations); thus, they can potentially report a broader variety of API misuses, but with weaker guarantees of precision compared to our experiments. In addition, there is a natural trade off between breadth of detected misuses and how easily addressable they are; our focus on exception preconditions can lead to actionable (and possibly even automatic) code improvement suggestions.

V. Conclusions and Future Work

Two main directions to mature this paper's idea of analyzing exception preconditions of library methods in client code to suggest code improvements are: 1) improving the quality and quantity of exception preconditions; and 2) automating the generation of code improvement suggestions. Direction 1) is motivated by findings that a small portion of a library is generally responsible for a large portion of client usage [8], [11]; and can naturally lead to progress in direction 2) by specializing the suggestions to cover the most common cases. Finally, applying our ideas to different benchmark collections [1], [10] is also a natural way to further validate them.

Dataset: the complete dataset of our experiments is available: https://doi.org/10.6084/m9.figshare.23634747.

References

[1] Sven Amann, Sarah Nadi, Hoan Anh Nguyen, Tien N. Nguyen, and Mira Mezini. MUBench: a benchmark for API-Misuse Detectors. In *MSR*, pages 464–467. ACM, 2016.

[2] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Trans. Software Eng.*, 45(12):1170–1188, 2019.

[3] K. Bourrillion. Nulness Design FAQ. https://github.com/jspecify/jspecify/wiki/nullness-design-FAQ, 2022.

[4] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *MSR*, pages 560–564. IEEE/ACM, 2021.

[5] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming. SANER, 2017.

[6] Thomas Durieux, César Soto-Valero, and Benoit Baudry. Duets: A dataset of reproducible pairs of Java library-clients. In *MSR*, pages 545–549. IEEE, 2021.

[7] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. Usability-oriented design of liquid types for java. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1520–1532. IEEE, 2023. doi:10.1109/ICSE48619.2023.00132.

[8] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. API beauty is in the eye of the clients: 2.2 million maven dependencies reveal the spectrum of client-API usages. *J. Syst. Softw.*, 184:111134, 2022.

[9] C. A. R. Hoare. Null references: The billion dollar mistake. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/, 2009.

[10] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. Evaluating Automatic Program Repair Capabilities to Repair API Misuses. *IEEE Trans. Software Eng.*, 48(7):2658–2679, 2022.

[11] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. A Large-scale Study on API Misuses in the Wild. In *ICST*, pages 241–252. IEEE, 2021.

[12] Diego Marcilio and Carlo A. Furia. How Java programmers test exceptional behavior. In *MSR*, pages 207–218. IEEE, 2021.

[13] Diego Marcilio and Carlo A. Furia. What is thrown? Lightweight precise automatic extraction of exception preconditions in Java methods. In *ICSME*, pages 340–351. IEEE, 2022.

[14] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P. Robillard. Generating unit tests for documentation. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi:10.1109/TSE.2021.3087087.

[15] Hoan Anh Nguyen, Hung Dang Phan, Syeda Khairunnesa Samantha, Son Nguyen, Aashish Yadavally, Shaohua Wang, Hridesh Rajan, and Tien N. Nguyen. A hybrid approach for inference between behavioral exception API documentation and implementations, and its applications. In *ASE*, pages 2:1–2:13. ACM, 2022.

[16] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212. ACM, 2008.

[17] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. Understanding the exception handling strategies of Java libraries: an empirical study. In *MSR*, pages 212–222. ACM, 2016.

[18] Hushuang Zeng, Jingxin Chen, Beijun Shen, and Hao Zhong. Mining API constraints from library and client to detect API misuses. In *APSEC*, pages 161–170. IEEE, 2021.

[19] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. An empirical study on API parameter rules. In *ICSE*, pages 899–911. ACM, 2020.

[20] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation. *IEEE Trans. Software Eng.*, 46(9):1004–1023, 2020.

URL References

a. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Random.java#L383-L388

b. https://github.com/AlphaAutoLeak/zelix-injection/blob/master/src/main/java/zelix/utils/Utils.java#L256

c. JavaParser: https://github.com/javaparser/javaparser

d. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/StringBuilder.java#L228C40-L228C40

e. https://github.com/feathersui/feathersui-starling-sdk/blob/master/modules/swfutils/src/java/flash/swf/tools/SwfxParser.java#L173

f. https://docs.oracle.com/en/java/javase/20/jshell/introduction-jshell.html

g. https://github.com/aaiyer/SuanShu/blob/ed9829aed161112e4d5fb5e2a1ab5ae05d99a491/src/main/java/com/numericalmethod/suanshu/vector/doubles/dense/operation/Basis.java#L83

h. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/concurrent/LinkedBlockingDeque.java#L182

i. https://github.com/microsphere-projects/microsphere-java/blob/main/microsphere-core/src/main/java/io/microsphere/convert/multiple/StringToBlockingDequeConverter.java#L31

j. https://github.com/msdeep14/getAheadWithMe/blob/main/LowLevelDesign/Concurrency/src/practice/ratelimiter/strategy/LeakyBucket.java#L15

k. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/ArrayDeque.java#L194

l. https://jspecify.dev/

m. https://github.com/jfree/jfreechart/blob/5aac9ae42147d34fe175e29af3993172e9c9080a/src/main/java/org/jfree/chart/JFreeChart.java#L257

n. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Properties.java#L224

o. https://github.com/shtrih-m/javapos_shtrih/blob/master/Source/Core/src/com/shtrih/util/Localizer.java#L169

p. https://github.com/jenkinsci/pipeline-groovy-lib-plugin/blob/773332a145baaa64a936eb23019e92dc110f7bc0/src/main/java/org/jenkinsci/plugins/workflow/libs/LibraryAdder.java#L172C5-L172C5

q. https://fbinfer.com/docs/all-issue-types/#nullptr_dereference

r. https://scan.coverity.com/

s. https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html

t. https://rules.sonarsource.com/java/RSPEC-2259/

u. https://rules.sonarsource.com/java/RSPEC-3655/

v. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Optional.java#L148