2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

# Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study

Demóstenes Sena[1, 2], Roberta Coelho[1], Uirá Kulesza[1], Rodrigo Bonifácio[3]

[1]Informatics and Applied Mathematics Department (DIMAp), Federal University of Rio Grande do Norte, Brazil
demostenes.sena@ifrn.edu.br, roberta@dimap.ufrn.br, uira@dimap.ufrn.br

[2]Federal Institute of Education, Science and Technology of Rio Grande do Norte, Brazil
[3]Department of Computer Science, University of Brasília, Brazil
rbonifacio@unb.br

## ABSTRACT

This paper presents an empirical study whose goal was to investigate the exception handling strategies adopted by Java libraries and their potential impact on the client applications. In this study, exception flow analysis was used in combination with manual inspections in order: (i) to characterize the exception handling strategies of existing Java libraries from the perspective of their users; and (ii) to identify exception handling anti-patterns. We extended an existing static analysis tool to reason about exception flows and handler actions of 656 Java libraries selected from 145 categories in the Maven Central Repository. The study findings suggest a current trend of a high number of undocumented API runtime exceptions (i.e., @throws in Javadoc) and Unintended Handler problem. Moreover, we could also identify a considerable number of occurrences of exception handling anti-patterns (e.g. Catch and Ignore). Finally, we have also analyzed 647 bug issues of the 7 most popular libraries and identified that 20.71% of the reports are defects related to the problems of the exception strategies and anti-patterns identified in our study. The results of this study point to the need of tools to better understand and document the exception handling behavior of libraries.

## CCS Concepts

**• Software and its engineering – Software creation and management – Software development techniques – Error handling and recovery.**

## Keywords

Exception handling; Exception flows analysis; Exception handling anti-patterns; Software libraries; Empirical study; Static analysis tool.

## 1. INTRODUCTION

The (re)use of software libraries or application programming interfaces (APIs) is a widespread approach for reducing software development time and cost. Hardly any modern application is developed nowadays without using at least a set of external libraries. Although software libraries and APIs often boost software development productivity, they also bring threats to application dependability. In particular, the lack of detailed documentation [6], insufficient testing [6] [27] [21], late-discovered bugs [6], backward compatibility problems [26] [32] [2], and uncaught exceptions [15] have been pointed out as some of the threats related to libraries reuse.


*MSR'16*, May 14-15, 2016, Austin, TX, USA
 ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901757

Modern applications have to deal with an increasing number of exceptional conditions as well as their reused libraries. The way a software library deals with exceptional conditions (i.e., its approach for signaling and handling exceptions) may directly affect the robustness of the client application. For instance, an undocumented runtime exception signaled by a library may cause an application crash; moreover, if the library silences an exception (which should report an unrecoverable situation), it may lead the application to an inconsistent state.

This work presents an empirical study whose goal was twofold: (i) to characterize the exception handling strategies of existing Java libraries from the perspective of their users; and (ii) to identify exception handling anti-patterns present in such libraries. Overall, the exception handling strategies of 656 Java libraries selected from 145 categories in the Maven Central Repository were investigated. In addition to that, we have also analyzed 647 bug issues of the 7 most popular libraries and identified that 20.71% of the reports are related to the problems of the exception strategies and anti-patterns identified in our study.

The study was based on exception flow analysis and manual inspections. We extended an existing tool (PLEA) [11] to support the exception analysis of libraries. The following outcomes were consistently detected in this study:

- A high number of the studied libraries (76.52%) does not document their API runtime exceptions (in at least one of the analyzed exception flows).
- The subsumption of exceptions was detected in 63.64% of all exception flows caught within the investigated libraries. This number shows a high potential of Unintended Handler Action problem [17] in such flows.
- Exception handling anti-patterns were detected in 25% (on average) of the exception flows of the investigated libraries.
- Considering only the flows on which an anti-pattern was detected, the *Catch and Ignore* (also known as exception swallowing) occurred in 30.38% of all flows.
- Inspecting the issue tracker systems of the 7 most popular libraries revealed that, on average, 20.71% of the issues were related to exception handling bugs.

Our findings point to threats not only to the development of libraries, but also to any robust application. The study results are relevant to Java developers who may underestimate the effect of exception handling anti-patterns and who may face the difficulty of

preventing them. Moreover, such findings call for improvements on the exception handling documentation of existing libraries.

The remainder of this paper is organized as follows. Section 2 provides the main concepts associated with our study. Section 3 presents the study design. Section 4 reports the study results. Section 5 provides a discussion of the wider implications of our results. Section 6 presents the threats to validity associated to this study. Finally, Section 7 describes related work, and Section 8 concludes the paper and outlines directions for future work.

# 2. TERMINOLOGY

This section presents the exception handling terminology used along this paper.

**Exception Type**: In Java, exceptions are represented according to a class hierarchy, in which every exception is an instance of the Throwable class, and can be of three kinds [14]: (i) checked exceptions (extends Exception), runtime exceptions (extends RuntimeException) and errors (extends Error). Checked exceptions received their name because they must be declared in the method's signature and the compiler statically checks if appropriate handlers are provided within the system. Both runtime exceptions and errors are also known as "unchecked exceptions", as they do not need to be specified in the method exception interface and do not trigger any compile time checking.

**Handler Type**: Each handler (i.e., catch clause) is associated with an exception type, which specifies its handling capabilities – which exceptions it can handle. The representation of exceptions in type hierarchies allows type *subsumption* to occur: when an exception is handled, it can be subsumed into the type associated to a handler, if the exception type associated to the handler (i.e., the handler type) is a supertype of the exception type being caught.

**Handler Action** comprises the statements inside the catch clause responsible for performing any recovery action.

**Exception Flow** is a path in a program call graph that links the method that throws the exception (i.e., the method signaler) and the method that handles it (i.e., the method that contains the catch clause). Notice that, if there is no handler for a given exception inside the program boundary, the exception path starts from the signaler and finishes at a program or library entry points.

**Internal Exception** is an exception signaled and handled within a component/library – the library invokes its own internal handlers in response to such exceptions.

**API Exceptions** are exceptions signaled within a component/library but the handlers responsible for dealing with such exceptions are external to the signaling library. In other words, such exceptions cross the boundary of the library and its clients are the ones responsible for implementing the handler strategy.

**Exception Documentation** comprises the information regarding the exceptions that may be thrown by a method – such information can be part of the method signature (in the throws clause) or in the corresponding Javadoc (using the *@throws Javadoc annotation*). Such documentation alerts the method's users of the existence of such exceptions, so they can define proper handlers to them.

Therefore, the **exception handling strategy** of a library comprises all decisions regarding its exception handling code, which include: (i) the exception types that should be signaled inside the library; (ii) the exceptions that should be handled internally and the handler actions that should be taken; and (iii) the exceptions that should cross the program boundary and therefore should be part of the library API.

# 3. STUDY SETTINGS

This section describes the study settings by presenting details about the study goals, research questions, target libraries, and the procedures we take for both data collection and data analysis.

## 3.1 Study Goals and Research Questions

Our study was conducted by means of a deeper analysis of the exception flows and handler actions used in Java libraries. We aimed at revealing the impact of the adopted exception handling strategies from the perspective of libraries' users. In order to address such aim, the following research questions (RQs) guided our study:

*RQ 1. What are the characteristics of the exception flows found in Java libraries?*

We answered this research question by collecting and examining the exception flows from existing Java libraries. We collected characteristics of exception flows such as: the exception type (e.g., checked or runtime), the handler type (e.g., specific handler or subsumption), the exception boundary (e.g., internal or API exception); and the existence of exception documentation. The lack of exception documentation is particularly harmful in API methods since the client cannot foresee the exceptions that may come from it and hence cannot adequately implement the code to deal with them.

*RQ 2. Which are the handler actions implemented in Java libraries?*

This research question complements the first one. We addressed this research question by examining handler actions implemented for each exception flow found in Java libraries. These handler actions can also reveal inadequate and potentially harmful exception handling strategies.

*RQ 3: What is the impact of exception handling strategies on the reported bugs?*

This research question was defined to reveal if the exception handling strategies of libraries could be the cause of bugs reported by a library or clients developers. We inspected the issues from issue tracker systems to find out which ones were caused by the exception handling code. We also relate such bugs with the exception handling strategies mined on the previous research questions.

## 3.2 Target Java Libraries

In order to conduct our study, we selected the three most popular libraries and their dependencies from each 145 different categories available in Maven Central Repository (http://search.maven.org/), overall they comprised 656 Java libraries. Table 1 shows some metrics regarding the source code and exception flows of these libraries. We decided to use some metrics already discussed in a previous study [10] to classify our collected libraries. According to the number of lines of code (KLoC), the libraries have been classified as Small (… - 20,000 LoC), Medium (20,001 - 100,000 LoC) or Large (100,001+ LoC). The analyzed libraries have about 23KLoC (on average) – overall they contain approximately 15,116KLoC. We have analyzed 28,819 exception flows – on average 43 flows per lib.

Furthermore, we manually inspected the issues of the seven most popular Java Libraries according to the Maven Central Repository. The inspected libraries with their respective size in KLoC were: Commons-collections (CC) – 64KLoC, Commons-lang (CL) with 56KLoc, Log4J (LJ) – 44KLoC, Maven-filtering (MF) – 37KLoC,

Plexus-utils (PU) – 32KLoC; Easymock (EM) – 11KLoC, and Slf4j-api (SA) – 4 KLoC.

**Table 1: General information about libraries.**

| | |
|---|---|
| # Libraries | 656 |
| # Small libraries | 485 |
| # Medium libraries | 144 |
| # Large libraries | 27 |
| # of LoC | 15,116,910 |
| # of LoC (mean) | 23,044.07 |
| # of LoC (median) | 7,693.50 |
| # of flows per lib (mean) | 43.46757 |
| # of flows per lib (median) | 29 |
| # of flows per 100KLoC (mean) | 2,431.407 |
| # of flows per 100KLoC (median) | 413.6814 |
| # of flows | 28,819 |

## 3.3 Exception Flows

We adapted PLEA tool [11] to collect the exception flows of each analyzed library (see Section 3.5). Then we distilled the information embedded in each flow, such as, the methods responsible for raising (aka the signaler method) and catching (aka handler method) exceptions. For each of these methods, we obtained the meaningful information for our analysis (e.g., method signature, exception thrown, and catch argument). Furthermore, we identified both the procedures taken by the library developers to handle a given exception as well the situations that characterize uncaught exceptions.

Initially, we classified the exception flows according to the existence of a catch block – the flows represent either a caught exception or an uncaught exception. Additionally, we mined the handler type (i.e., specific or subsumption). For the uncaught exceptions, we classified them according to the exception type (i.e., checked or runtime). Hence, if the propagated exception is runtime, we categorized it as an API runtime exception, otherwise, as an API checked exception.

In cases where the exception is an API exception (crosses the library boundary) and was also a runtime exception, we performed two particular analyses to acquire relevant information. Firstly, we examined the Javadoc annotations (i.e., *@throws*) library API methods. The goal of this analysis was to verify if the developers had reported the API runtime exceptions. For the exception caught within the library, we classified the handler actions according to an existing classification detailed in previous works [1] [22]. Some handler actions identified in these studies were not considered because the implementation of the identification heuristic for them would be very complex, which extrapolates the scope of our study (e.g. rollback). Table 2 illustrates the handler actions with short descriptions.

Despite the descriptions of handler actions shown in Table 2, some of them can be implemented as a combination between two or more handler actions. For example, *commons-dbcp* (release 1.4) library has a catch block that is implemented by *method(e)* and *return* statements where *e* is the caught exception object.

## 3.4 Bugs related to Exceptions

For the issues analysis, a set of exception bugs was analyzed and classified according to their causes. The decision about exception bug causes was detected by analyzing the problem description and discussions fields posted for each bug issue by users or developers of the libraries.

**Table 2: Handler action types.**

| Handler Action | Description |
|---|---|
| *throw new(e)* | Create a new exception type using the caught exception. |
| *throw e* | Re-throw the caught exception. |
| *log(e)* | Call logging method using the caught exception as a parameter. |
| *Log(..)* | Call logging method. |
| *Log(e.message)* | Call logging method using the caught exception message as a parameter. |
| *method(e)* | Call a method using the caught exception as a parameter. |
| *method(..)* | Call a method. |
| *return* | Return statement. |
| *empty* | Empty catch block. |

In this section, we present the results of the manual analysis of the issue tracker systems of the 7 most popular libraries from Maven Central Repository. We have manually identified the issues directly related to exception handling strategies, such as undocumented API runtime exceptions and anti-patterns (e.g., destructive wrapping), or indirectly related to them, for example, increasing optimization due to replace a throw statement by and assertion statement. The main aim was to understand if the collected results of the exception strategies in RQ1 and RQ2 are related to bugs reported by users of the libraries.

We performed a keyword search using the following queries: (a) JIRA: *description ~ "exception"* and (b) GitHub: *inbody is:issues "exception"*). After that, we identified if the reported bug was related to exception and classified it according to its causes. The observed exception bug causes are related to exception anti-patterns [30], Javadoc and/or other specific exception handling problems.

## 3.5 Procedures and Measures

As mentioned before, the information about exception flows was collected using PLEA – an existing tool developed by our research group [11]. PLEA was initially designed to collect exception flows in software product lines (SPL). It performs an analysis of SPLs as a Java application to collect the exceptions flow data (signaler and handler methods). After that, it identifies the feature expression related to each of them.

In this study, we extended the PLEA tool to analyze Java libraries. The main difference between the analysis for SPL (or Java applications) and libraries is the notion of entrypoints. The entrypoints of Java applications are the main methods (i.e., *main (String [])*), while the entrypoints of Java libraries are the public API methods. In addition, we also extended the PLEA tool to identify handler actions for caught exception flows and Javadoc annotations in the entrypoints (i.e. public methods) that propagate some exception. The main goal to collect the Javadoc annotations (*@throws*) in the entrypoints is to identify the documented API runtime exception. The documentation is a strategy to indicate to the libraries' users that some runtime exception might be propagated [13].

Figure 1 gives an overview of our study. First, a set of libraries from Maven Central Repository was automatically recovered (1). For each recovered library, a subset of entrypoints (sample) was selected (2), so that, due to memory constraints, make the analysis feasible (see Section 6 – Threats to Validity). We applied the Yamane's formula [31] (10% confidence level) combined with the Reservoir Algorithm [4] to determine the amount of sample and which libraries' entrypoints should be selected. Then, the call graph

was built from the bytecode and the selected entrypoints' (3) applying the 1-CFA algorithm [18] provided by WALA framework [33]. These call graphs were represented by an internal structure of the WALA framework and they were traversed to discover the respective exception flows (4). The next step discovers the handler actions and documentations (i.e. *@throws* Javadoc annotations) related to each exception flow (5). For the handler action identification, the source code of handler method related to each exception flow was automatically analyzed using a developed heuristic. This heuristic was implemented using a Java Parser and it was based on a previous manual analysis performed in three hundred (300) catch blocks of the eight most popular Java libraries from the Maven Central Repository. This documentation analysis was performed in the source code of entrypoints (public API methods) that propagate some runtime exception. The component responsible for this analysis navigates on the source code to find @throws Javadoc annotation in these methods. Before that, it linked the documented exception types to the respective public methods. Thus, it is possible to compare the API runtime exceptions indicated by the exception flow analysis to the documented exception types for each entrypoint. Finally, this process produced a developed representation of the exception flows and their handler actions, which were inserted into a local relational database (6). Thus, researchers can run SQL scripts to collect particular information about the exception flows (7), such as a query to obtain all exception flows that have empty handler actions (i.e. swallowing the caught exception) can be easily specified.

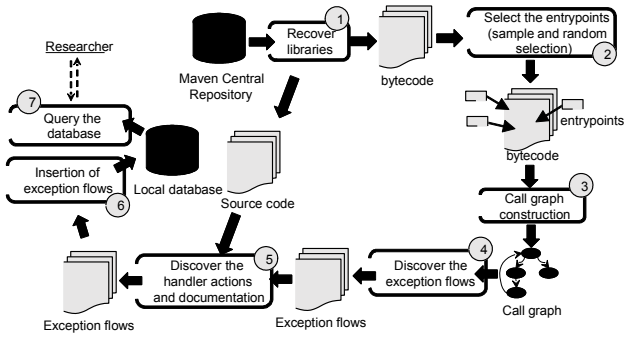


**Figure 1: Study Overview.**

For the issues analysis, we collected the reported issues of JIRA repositories of CC, CL, LJ, MF, PU and SA libraries (see Section 3.2), and ones of GitHub repository of EM for issues analysis.

# 4. STUDY RESULTS

In this section, we present and discuss the results of our study. All the data collected and generated in the study can be found elsewhere [7].

## 4.1 RQ 1: What are the characteristics of the exception flows found in Java libraries?

As the starting point of our analysis, we calculated the number of API checked or unchecked exceptions. Then for each exception handled within the library, we observed whether the handler type follows the subsumption or specific strategy. Our intention was to evaluate whether this information could reveal violations to exception handling recommended practices [23] and Java specification [14]. For instance, Subsumption may cause the Unintended Handler Action problem [17] that occurs when a developer does not know the incoming exception and therefore he/she might apply inadequate handler actions to deal with the cause of error. API runtime exceptions are also a potential source of bugs, especially when they are not documented in the API methods, and consequently the developers do not know about them. On the other hand, the use of specific handlers and checked API exceptions follows the recommendations. Specific handlers provide the opportunity for developers to implement suitable handler actions. When checked exceptions remain uncaught, compilers and IDE tools statically check this kind of violation to the Java language specification and alert the developers – which allow them to figure out API checked exceptions.

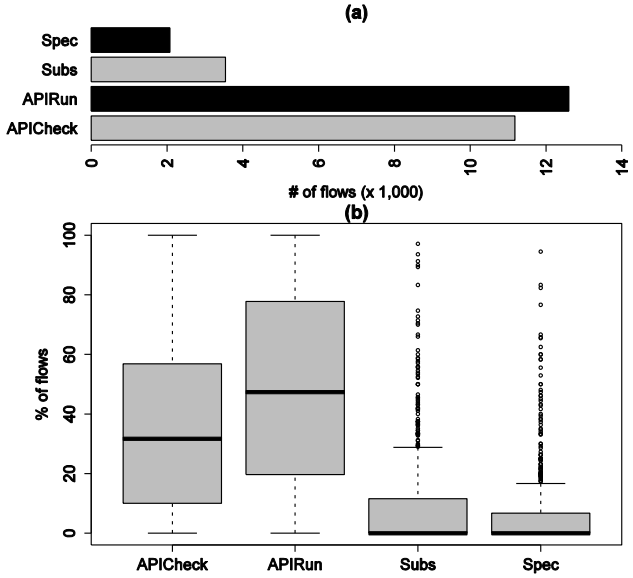


**Figure 2: Absolute Numbers and Distribution of Exception Flows.**

Figure 2 shows the absolute numbers and distribution of exception flows according to their classification. Figure 2a shows that the number of API runtime exceptions (APIRun) is greater than the number of API checked exceptions (APICheck), exceptions caught by subsumption (Subs), and caught by specific handlers (Spec). We can also observe that the number of API exceptions – the ones that remain uncaught within the library – is greater than caught exceptions.

The boxplots of the Figure 2b reinforce the previous information. It indicates that the median of API runtime exceptions (47.4%) per library is greater than median of API checked (31.71%), exceptions caught by subsumption (about 0.0%), and specific handlers (0.0%). These results suggest that API runtime exceptions are prevalent among the exception flows, and the number of caught exception flows (subsumption and specific) is much smaller compared to uncaught exceptions for the analyzed Java libraries.

**The range of API runtime exceptions is from 19.67% to 77.78%, API checked exceptions is from 10% to 56.83%, subsumption is from 0% to 11.52% and specifics is from 0% to 6.67%.**

**Undocumented API Runtime Exceptions**. Despite the prevalent number of API runtime exceptions from the analyzed Java libraries, it does not represent a critical problem or a bad practice if the API runtime exceptions are documented [17]. An API runtime exception is documented when the correspondent API method has a specific Javadoc annotation detailing the possible exceptions that might be thrown. Hence, we investigated the Javadoc annotations

*(@throws*) of API methods in the libraries that have at least one API runtime exception, totaling 618 Java libraries. This investigation aims at identifying the number of documented API runtime exception using Javadoc annotations.

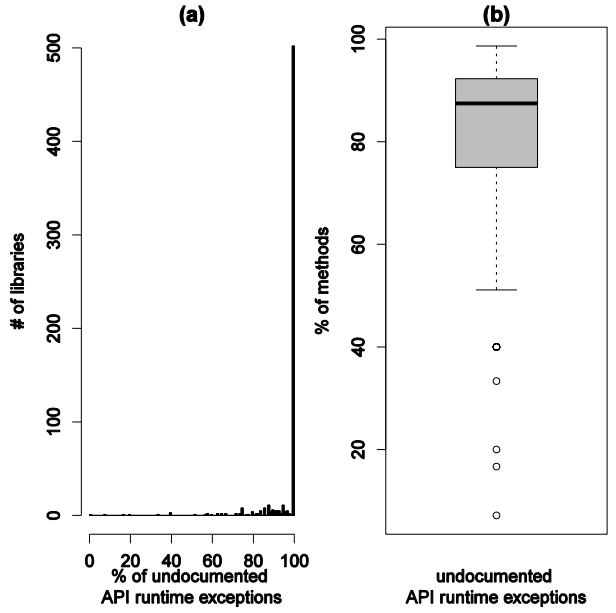


**Figure 3: (a) Distribution and (b) dispersion of undocumented API runtime exceptions.**

Figure 3 shows the distribution (Figure 3a) and the dispersion (Figure 3b) of undocumented API runtime exceptions for the analyzed libraries. Figure 3a shows the distribution of libraries according to their percentage of undocumented API runtime exceptions. Figure 3a illustrates that most of the libraries do not comply with the recommendation of documenting API runtime exceptions using Javadoc annotations, that is, 502 libraries propagate at least one undocumented API runtime exception. We plotted the dispersion (Figure 3b) considering only the 116 libraries that have at least one documented API runtime exception. Figure 3b shows that the majority of libraries have their results varying from 75% to 92.31%. These numbers suggest a small amount of documented API runtime exceptions for the analyzed Java libraries. As a concrete example, the commons-math (release 2.2) library has 87.81% of their API runtime exceptions undocumented. This library has 31 API runtime exceptions and only 4 of them have related Javadoc annotations.

**The amount of analyzed libraries that do not have documented API runtime exceptions is 502 (76.52%). Only a small amount of the quantified API runtime exceptions – varying from 7.6% to 20.6% - is documented using Javadoc annotations.**

**Potential Unintended Handler Action**. Unintended exception handler is a harmful behavior that can decrease the dependability of systems reusing existing libraries [24]. Exception subsumption is a possible source of the Unintended Exception Handler problem. For characterizing the studied libraries, we have collected the number of exception flows that were implemented using subsumption handler.

Considering all exception flows as shown in Figure 2, subsumption and specific exception flows vary from 0.0% to 11.5% and from 0.0% to 6.7%, respectively. Since our work is related to the handler types, our analysis focused on libraries that have at least one caught exception flow (418 libraries).

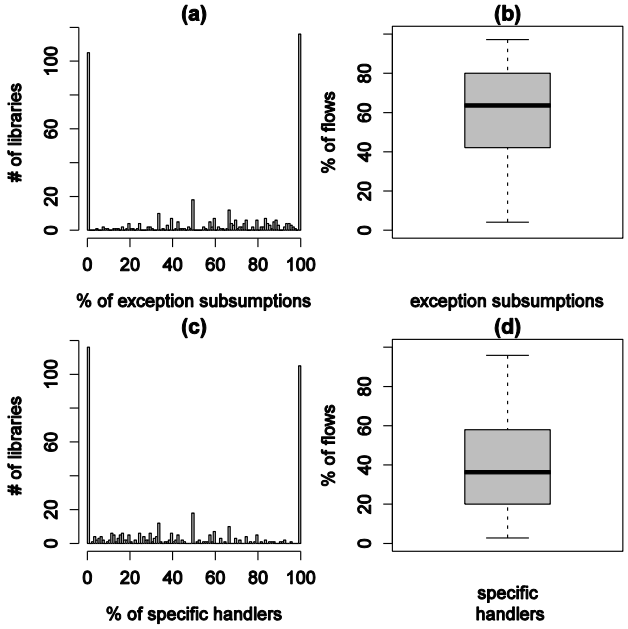


**Figure 4: Subsumption and Specific Exceptions Flows.**

Figure 4a and Figure 4c show the histogram of caught exception flows. According to our analysis, the libraries can be grouped into three categories according to the percentage of caught handler type: (a) 116 libraries only have exceptions caught by subsumption (Figure 4a), (b) 105 libraries only have specific handlers (Figure 4c); and (c) 197 libraries have at least one exception subsumption and one specific handler (Figure 4a and Figure 4c). Furthermore, we plotted the dispersion diagrams of the percentage of subsumption (Figure 4b) and specific (Figure 4d) flows examining only the libraries in the third group because we can identify the profile of caught exception flows. Figure 4b indicates that the majority of third group libraries have their subsumption exceptions varying from 42.1% to 80%. These values are greater than the values observed of the specific exceptions (Figure 4d) that correspond from 20.0% to 57.9%.

These results also indicate that the median of subsumption is greater than specific exceptions for the 197 analyzed libraries. It can be reinforced by showing that the amount of libraries (116) that have only subsumption exceptions is greater than the libraries (105) that have only specific exceptions.

**The number of libraries that have only subsumption flows (27.75% of the 418 libraries that have caught exceptions) is greater than the ones that only have specific flows (25.12%). The range of subsumption varies from 42.11% to 80%, and specific varies from 20% to 57.89% in the libraries that have at least one caught exception (47.13%).**

**Too General Catch Clauses**. A more critical scenario of the Unintended Handler Action problem [17] happens when a handler catches checked and runtime exceptions using more-general types (e.g. Exception or Throwable) as catch argument. This scenario was defined as an anti-pattern [30] because it can bring problems to the exception handling through the inadequate capture of specific exceptions by means of a generic handler. Hence, we examined the

subsumption exception flows to find out how many of them can capture checked and runtime exceptions.

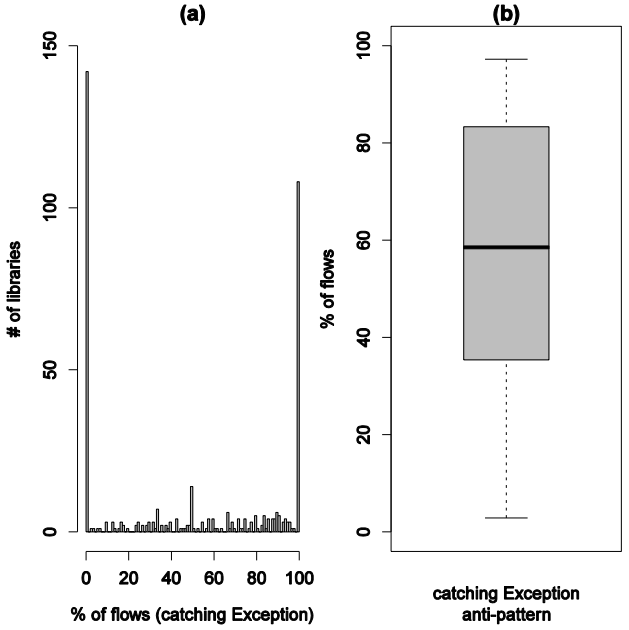


**Figure 5: Histogram and Boxplot of Catching Exception Anti-pattern.**

Figure 5a shows the histogram for this analysis. We can observe that 108 (25.84%) libraries only catch Exception flows, 142 (33.97%) libraries do not catch Exception flows, and, finally, 168 (40.19%) libraries catch at least one Exception flow. We examined the caught exception flows of the last library category to characterize the profile of this anti-pattern. Figure 5b shows the dispersion of caught exception flows of the libraries that have some Catching Exception flow. We can observe that the amount of Catching Exception flows in analyzed libraries varies from 35.4% to 83.3% of caught exception flows.

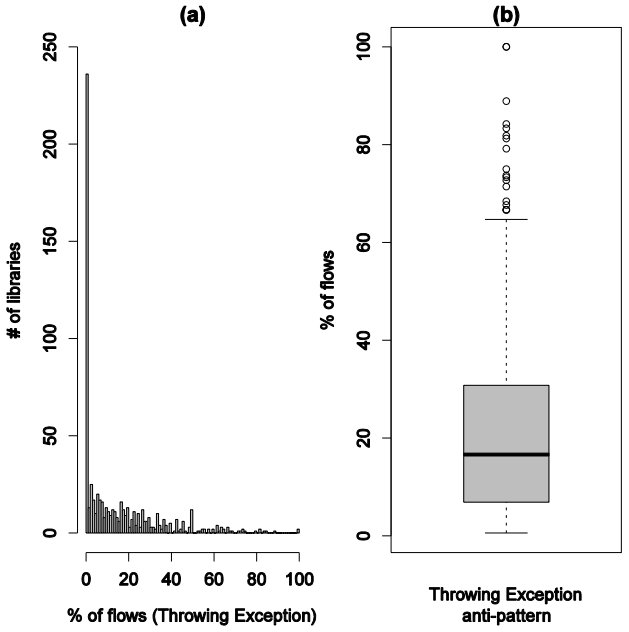


**Figure 6: Dispersion of Throwing Exception Anti-pattern.**

**The amount of libraries that have only Catching Exception flows (25.84%) and the range (from 35.39% to 83.33%) of flows that implement Catching Exception indicate a high global number of exception flows that implement this anti-pattern.**

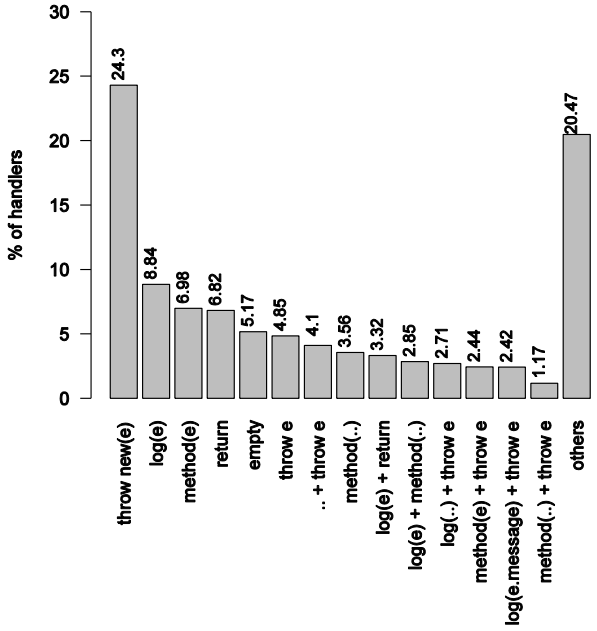


**Figure 7: Percentage of handlers that implement some type of handler action combination.**

**Too General Throw Statements.** We analyzed the 656 libraries and collected the distribution of libraries according to the amount of exception flows that implement the Throwing Exception anti-pattern (Figure 6a). We observed that 434 (66.16%) of libraries use the Throwing Exception anti-pattern, and most of these exception flows vary from 6.90% to 30.77% (Figure 6b).

## 4.2 RQ 2: Which are the handler actions implemented in Java libraries?

In this section, we present the results related to the implementation of handler actions in the caught exception flows from the analyzed Java libraries.

Initially, we examined and classified the existing handler actions of the caught exception flows. We have used a previously published classification [1] [22]. Figure 7 shows the percentage of catch blocks for each type of handler action according to the classification shown in Table 2. For example, 3.32% of the handlers implement the combination of *log(e)* and return statement. We can observe that *throw new(e)*, *log(e)*, *call method(e)*, *return* and *empty* statements were the most common handler actions found in our study, respectively, 24.30%, 8.84%, 6.98%, 6.82 and 5.17% percentage of handlers.

After the initial classification of handler actions, we examined them in the light of exception handling anti-patterns reported by the community [30]. Our main purpose was to understand the exception handling strategies of the analyzed libraries. Figure 8a shows the distribution of libraries according to the number of caught exception flows that implement some anti-pattern. We can observe that 125 libraries (19.05%) do not implement any anti-pattern. Considering the libraries (80.95%) that have at least one implemented anti-pattern, Figure 8b exhibits the dispersion of

percentage of anti-pattern exception flows. We can observe that the majority of libraries have 10.7% to 45.7% of their exception flows related to some anti-pattern. These results indicate that a relevant number of the analyzed libraries have anti-patterns, and they correspond to a significant percentage of the global caught exception flows. An analysis of the collected results for different anti-patterns is presented next.

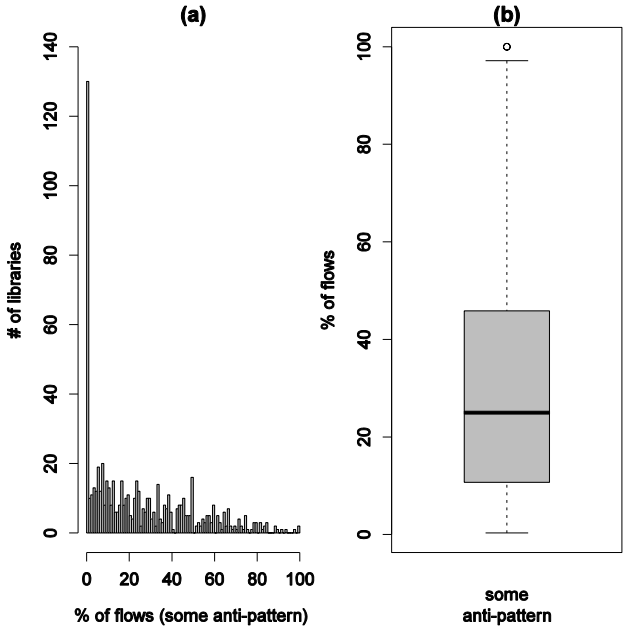


**Figure 8: Histogram and Boxplot of Java libraries that implement some anti-pattern.**

**19.05% from libraries do not implement any exception flow anti-pattern. Most of the libraries (80.95%) implement at least one anti-pattern. The percentage of exception flows that have anti-patterns varies from 10.71% to 45.83%.**

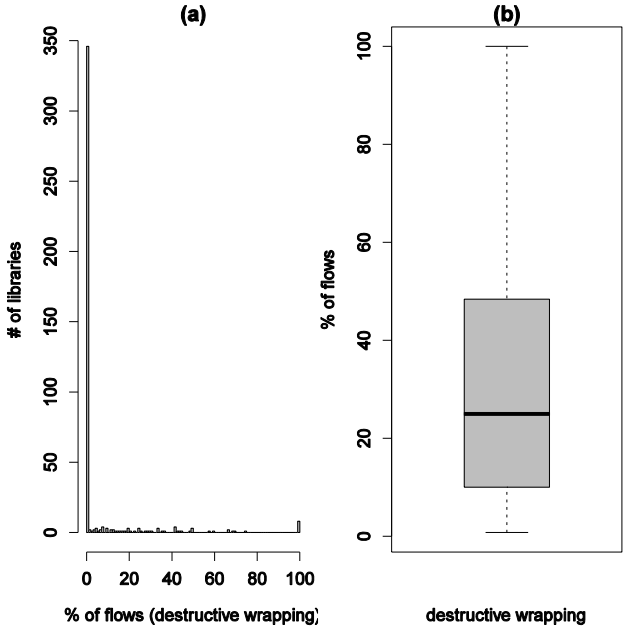


**Figure 9: Dispersion of Destructive Wrapping anti-pattern.**

**Catch and Ignore Anti-pattern**. In our study, we found a small number of occurrences of some anti-patterns. The complete list of the identified anti-patterns can be found elsewhere [7]. For example, the Destructive wrapping anti-pattern happened in 74 (17.70%) libraries (see Figure 9a) and the median of exception flows that implements this anti-pattern is 25% (Figure 9b).

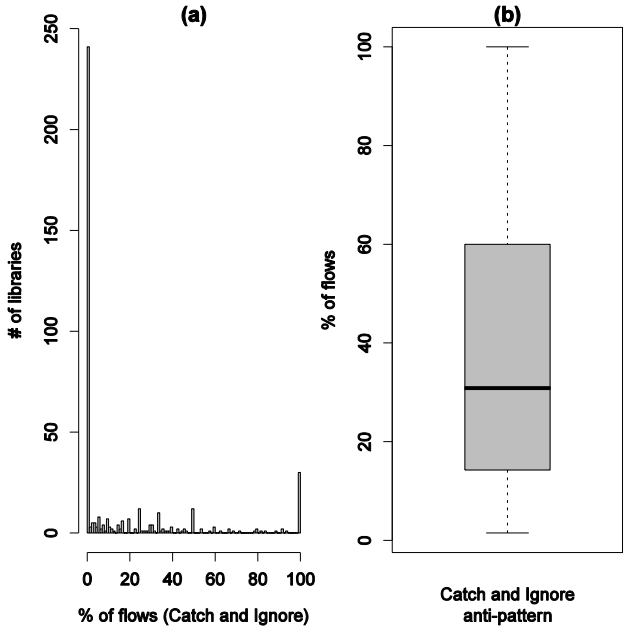


**Figure 10: Dispersion of Catch and Ignore anti-pattern.**

We have found different and more critical results for the Catch and Ignore anti-pattern. Figure 10a shows the distribution of libraries according the percentage of exception flows that implement Catch and Ignore anti-pattern. A total of 177 (42.34%) of libraries have exhibited the Catch and Ignore anti-pattern. The percentage of caught exception flows (Figure 10b) that implements this anti-pattern varies from 14.29% to 60.00% (median of 30.88%).

**Catch and Ignore anti-pattern happened in 177 (42.34%) of libraries and the range is from 14.29% to 60% of caught exception flows.**

Table 3 shows the top 5 libraries with the highest and lowest number of anti-patterns. It shows the number of exception handling elements (i.e., throw statements and catch blocks) and the number of anti-patterns per 100 EH elements. For example, mybatis library has 21 throw statements, 44 catch blocks and about 1,428 anti-pattern per 100 exception handling elements.

**Employed Statistical Tests**. We performed a statistical analysis to investigate possible correlations between the exception handling elements (i.e., throw statements and catch blocks) and the anti-patterns. We applied the Spearman's rank analysis in three scenarios of correlation: (i) number of caught exception flows and anti-patterns; (ii) number of throw statements and anti-patterns due to the throw statements; and (iii) number of catch blocks and anti-pattern due to the catch blocks. For a confidence level of 95%, the correlation results of this statistical analysis were 0.6648, 0.5056 and 0.7105, respectively, to the scenarios presented above. These results suggest that the libraries that have more anti-patterns implemented have more EH elements, mainly, catch blocks. It means that the number of catch blocks causes a high impact on the number of implemented anti-patterns, which was actually expected.

**Table 3: Top 5 libraries presenting the highest and lowest number of anti-patterns per 100 EH elements.**

| | Libraries | # of throw statements | # of catch blocks | # of anti-pattern per 100 EH elements | Categories |
|---|---|---|---|---|---|
| the lowest percentage | common-core (1.0.7) | 54 | 0 | 0.00 | Core Utilities |
| | high-scale-lib (1.1.1) | 52 | 0 | 0.00 | Collections |
| | ejml (0.25) | 47 | 0 | 0.00 | Vector/Matrix Libraries |
| | super-csv (2.1.0) | 42 | 0 | 0.00 | CSV Libraries |
| | objenesis (2.1) | 36 | 0 | 0.00 | Reflection Libraries |
| the highest percentage | aws-java-sdk (1.8.9.1) | 59 | 25 | 1,306.78 | Cloud Computing |
| | mybatis (3.2.5) | 21 | 44 | 1,428.57 | Object/Relational Mapping |
| | ehcache (1.2.3) | 99 | 24 | 1,520.83 | Cache Implementations |
| | c3p0 (0.9.1.2) | 76 | 72 | 2,234.21 | JDBC Pools |
| | krati (0.4.1) | 42 | 26 | 2,263.69 | - |

## 4.3 RQ 3: What is the impact of exception handling strategies on the reported bugs?

In this section, we present the results of the manual analysis of the issue tracker systems of the 7 most popular libraries from Maven Central Repository. We have manually identified the issues directly related to exception handling strategies, such as uncaught undocumented runtime exceptions and anti-patterns (e.g., destructive wrapping). The main aim was to understand if the collected results of the exception strategies in RQ1 and RQ2 are related to bugs reported by users of the libraries.

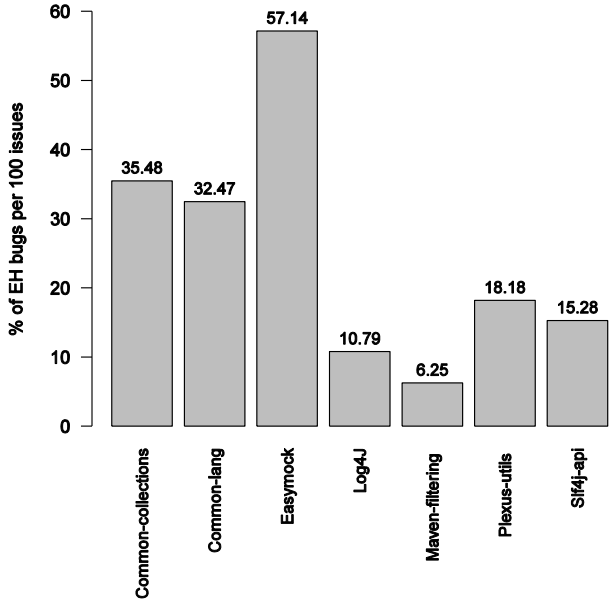


**Figure 11: Number the EH bugs per 100 analyzed issues for each library.**

Figure 11 presents the percentage of EH bugs for each studied library. We can observe that the numbers of EH bugs reported are significant, compared to the number of total bugs reported for each library. For example, CC, CL, EM have exhibited a total of35.48%, 32.47% and 57.14% EH bugs. Meanwhile, LJ, MF, PU and SA have presented inferior values varying between 6.25% and 18.18% EH bugs. However, in general terms, a total of 20.71% of the 647 analyzed EH bug issues are related to the problems identified in our study and reported in RQ1 and RQ2.

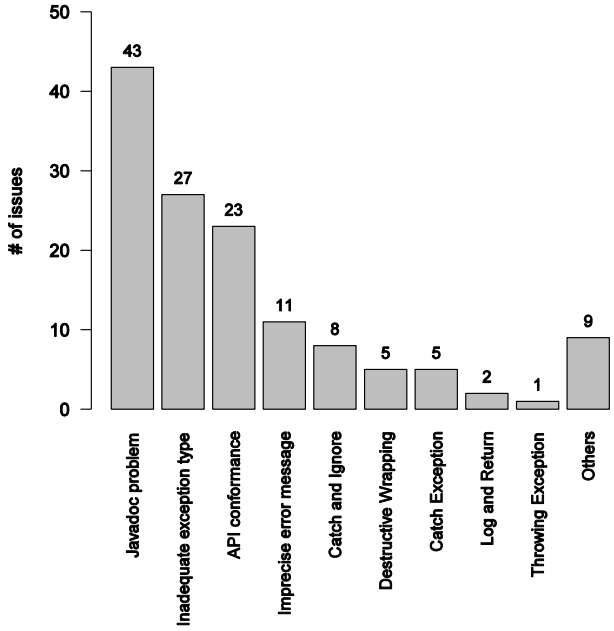


**Figure 12: Number of issues for each reported EH bug type.**

***Reported EH Bugs***. Figure 12 shows the type of EH bug identified and the number of occurrences on the collected issues. Javadoc problems (e.g., uncaught undocumented) was the most reported (43 out of 134 issues). By contrast, Throwing Exception anti-pattern had only one reported issue. We can observe that bugs that do not need more complex analysis (e.g., Javadoc problem, inadequate exception type, API conformance and imprecise error message) have a superior number of issues.

# 5. DISCUSSIONS AND LESSONS LEARNED

## 5.1 Undocumented API Runtime Exceptions

The high number of uncaught runtime exceptions found in this study supports the claims that library developers usually implement API methods that propagate runtime exceptions. This strategy delegates to applications developers the task of dealing with uncaught runtime exceptions. Hence, documenting the uncaught runtime exceptions might avoid certain kinds of runtime errors, since libraries' users will be aware about the uncaught runtime exceptions of a library as well as the cause of their occurrences.

This study found that library developers do not usually produce Javadoc annotations for uncaught runtime exceptions, which reinforces the importance of automated support to aid developers to identify and analyze uncaught runtime exceptions and their respective Javadoc annotations. In addition to that, such automated analysis might also provide a better comprehension of the uncaught runtime exception reasons. Consequently, library developers could provide explicit Javadoc annotations for API runtime exceptions.

The tool developed for this study represents an initial step in that direction.

## 5.2 Handling the Cause of Exception

*Unintended Handler Action* is a known problem that has been reported in exception handling empirical studies [24]. This problem can be analyzed from two perspectives according to its cause. The first perspective is generated when developers use a subsumption catch argument in a catch block. This scenario has been studied by many research work [1] [22] [20], though not extensively for Java libraries. We found an expressive number of libraries that use the subsumption strategy.

The second perspective arises when a method propagates a more general exception (i.e. using *throws* clause) than the signaled exception. Thus, the respective handler method will capture by subsumption an exception type even if it has a specific exception type as its catch argument. We found that 66.16% of the libraries considered in this study present at least one occurrence of this anti-pattern.

These *Unintended Handler Action* [17] perspectives emphasize the original exception handling proposal, which advises that handler actions should address the cause of exception. This advice refers to one of the best practices that recommends developers to produce specific handlers [23]. Thus, developers should understand the exception flows by examining their causes and consequently produce suitable actions for their handlers. Furthermore, the exception flow comprehension allows the developers to adequate the exceptional interfaces of their methods. Here we reported that several popular Java libraries do not follow recommendations related to this issue. One of our goals is to investigate the implications of Unintended Handle Actions in a future work.

## 5.3 Handler Action Identification

As part of our study for understanding exception handling strategies, we observed that some handler actions occur more frequently and now we discuss about their main consequences.

*Throw actions*. Considering combined and solo, Throw is the most used handler action, it was identified in 52.21% of caught exception flows (almost 25% of handler actions *throw new(e)*). This handler action is more complex to reasoning about because it creates an exceptional chain. An exceptional chain is composed by at least a couple of signaler and handler methods, which the first reached handler method has the throw statement. Approaches recommend that wrap actions should be applied to adequate the exception type according to the context of a specific layer or component [17]. However, throw actions may bring up undesirables side effects when developers do not understand the actions applied in exception chain [5].

*Call method (e) or method (...)*. These handler actions occurred in 24.20% of catch blocks from all libraries. Actually, they correspond to a widespread approach found over the libraries. Nevertheless, both handler actions require automated support that allows to find out which methods are related to exception handling and where they can happen.

*Catch and Ignore anti-pattern.* In this study, we observed that three of the analyzed anti-patterns – the Catch and Ignore, Catching Exception, Throwing Exception – have a significant number of occurrences. The Catch and Ignore is the only one related to the handler action analysis. It happens when no action is taken to deal with the cause of a caught exception. Empty blocks are an example of this Catch and Ignore anti-pattern. These occurrences suggest that the developers cannot address the cause of exception during the library implementation. However, it is also possible that they do not know that these inadequate recovery actions are happening. We found that almost 5% of the handler actions are empty.

*Exception handling anti-patterns*. Many exception handling anti-patterns did not exhibit a significant number of occurrences. We reported that 20% of the libraries do not implement any anti-patterns. However, when we consider the amount of exception flows that implement some anti-pattern this number become quite representative and varies from 10.71% to 45.83% of exception flows (25% of median). Consequently, each analyzed exception handling anti-pattern become a potential problem that should be addressed. The initial process to identify exception handling anti-patterns is to collect handler actions combinations. Our extension of PLEA tool makes this process feasible and we intend to evolve it for checking the instances of these anti-patterns.

## 5.4 Reported Bugs x Static Analysis Cost

We could observe that the developers reported a considerable number of EH bugs (20.71% on average - see Section 4.3). Most of such bugs represent scenarios on which an exception is raised but the method signaling it does not contain the corresponding Javadoc (32.09%). This kind of bug could be identified by a pure exception flow analysis. However, some of the reported EH bugs such as the Destructive Wrapping anti-pattern (3.73%) are more complex/costly to be automatically identified, since they require some level of data flow analysis while performing the exception flow analysis. If on the one hand, such EH bugs are more complex to be automatically identified, on the other hand only a few of them cause a visible failure that was worth to be mentioned by developers on bug reports.

## 5.5 Combined Analysis of Exception Flows and Handler Actions

The previous discussions claim for increasing comprehension about exception handling which should combine the analysis of exception flows and the respective handler actions. This combined analysis can become feasible with automated support and, in this sense, we have improved our tool to support these analysis. Beyond that, our tool can check customizable design rules for exception handling policies considering the common and specific issues of software systems, software product lines [19] and software libraries.

# 6. THREATS TO VALIDITY

*Internal Validity* – Our static analysis tool has some memory usage limitation when we apply a more precise algorithm (1-CFA) to create the call graph from libraries. This limitation is a known existing challenge [25] when static analysis is used in the context of interprocedural analysis. A possible solution is to create the call graph applying a less precise algorithm (RTA). However, we have chosen to apply a more precise algorithm to create the call graph from a sampling of API methods. In this sampling process, we used the Reservoir Sampling algorithm and the amount of entrypoints (i.e., API methods) was defined according to Yamane's formula with confidence interval of 10% of total API methods for each library. Further, we did not include the exceptions raised in third party libraries. However, our study investigated the impact of exception handling strategies for users of libraries, and thus we assume that the libraries are responsible for raising the exceptions and they should deal with the respective causes or at least notify the users when an API runtime exception can be propagated.

*External Validity* – We limited our analysis to a subset of open-source libraries available in the Maven Central Repository. We

cannot extend these results to commercial libraries because the companies can implement different exception handling strategies from the ones found in this study. Such threat is similar to other empirical studies, which also use open-source systems [1] [22] [9]. Regarding the threats to validity of issues analysis, the main goal was to observe if the libraries' users could notice some impact in their applications due to the exception handling of existing libraries. However, the developers of the analyzed libraries could be responsible for reporting some issues; thus, it could affect the relevance of our results. Based on this fact, we performed an identification of the responsible for reporting the issues and we identified that 84.62% of the analyzed issues were not reported by the developers of the analyzed libraries. Hence, this result alleviated the threats to validity of issues analysis.

*Construct Validity* – A heuristic was used to identify the handler actions from catch block instructions. This heuristic was implemented guided by the manual analysis of about 300 catch blocks of the 8 most popular libraries of Maven Central Repository. However, it is possible to find combinations of statements in catch blocks that did not occur in the set of analyzed catch blocks that guided our heuristic. Thus, the precision of this heuristic can be affected by the found catch blocks in analyzed libraries.

# 7. RELATED WORK

This section presents studies and approaches of exception handling and shows their weakness, strengths and association with our study.

## 7.1 Exception Analysis Studies

*Handler Action Analysis*. Cabral and Marques analysis [1] focuses on catch argument and its respective actions applied to recovery the related error. However, the recovery precision is threatened due to the cause indicated by the type of catch argument may cluster subtypes. In contradiction, the analysis is more precise when the exception flow is analyzed and the incoming exception is identified. The study investigated the actions responsible for exception handling in 32 Java and .NET systems. They identified that the developers did not adequately apply the exception handling mechanisms. Our study is complementary to that work, because we have performed a combined analysis of exception flows and handler actions to improve the precision of the analysis.

*Exception Flow Analysis*. Fu and Ryder [5] and Chang et al [3] propose exception flow analysis to gather exception chains from Java applications. Their analyses indicate the methods traversed by exception object from raising to catching that increase the understanding about exception handling strategy. Sinha et al [28] show an exception flow analysis that identifies a set of anti-patterns, for example, large distance between throw and catch. These studies are closer to ours because they also analyze aspects of exception flows. However, they can complement our study because we did not investigate the intermediate methods from exception flows. On the other hand, we have performed the examination of recovery actions.

## 7.2 Exception Handling Approaches

*Recommendation*. Rahman and Roy [16], Barbosa et al [8] and Thummalapenta and Xie [29] have proposed approaches for recommendation of recovery actions based on mining software repository analysis. They have linked the context of protected block (i.e., try) and catch argument type to recommend the more suitable handler action combination. However, these approaches can present a reduced precision because the recommendation procedure uses the catch argument instead of incoming exception type and they ignore the cause of exception.

*Visualization.* Shah et al [12], Robillard and Murphy [17], Fu and Ryder [5], and Chang et al [3] have defined approaches for the visualization of exception handling. They can increase the understanding about raising, propagation and catching exception for each exception flow. These approaches would provide a more complete view of exception handling if they had combined the analysis of exception flows and handler actions.

# 8. CONCLUSIONS AND FUTURE WORK

This paper presented an empirical study of exception handling strategies in Java libraries. A combined analysis of exception flows and handler actions was performed with the main aim to characterize and reveal potential problems related to the exception handling strategies adopted by Java libraries. The study involved an automated analysis of 656 Java libraries from Maven Central Repository. In addition, we performed a manual analysis of the exception bugs reported in the issues tracker systems of the 7 most popular of these libraries. We also extended an existing static analysis tool to address such analyses.

As a result of our study, we found: (i) a high percentage of undocumented API runtime exceptions in the Java libraries, which points out to a potential source of exception bug if some of these exceptions remain uncaught in the client application; (ii) a significant number of exception subsumptions in the Java libraries, which can provoke the Unintended Handler Action problem; (iii) some scenarios of undesirable exception handling derived from the handler actions (i.e. anti-patterns). For example, the process of catching an exception and raising another exception without maintaining the cause of the first exception (i.e. Destructive Wrapping anti-pattern); and, (iv) a relevant number of exception bugs caused by inadequate exception handling strategies of third-party libraries extracted through manual analysis from issues tracker systems, which shows that the problems identified in the exception handling strategies are real and significant sources of bugs in existing applications.

For the future work, we are currently improving our exception analysis tool to support the automated analysis and verification of design rules for exception handling policies considering the common and specific issues of software systems, software product lines and software libraries. The main idea is to improve the reasoning about exception handling strategies performing the combined analysis of exception flows and handler actions as we have conducted in this paper.

**Acknowledgements.** This work is partially supported by the National Institute of Science and Technology for Software Engineering (INES), CNPq and FACEPE, grants 573964/2008-4, 552645/2011-7, and APQ-1037-1.03/08, CNPq Universal grant 484209/2013-2, and CAPES/PROAP.

# 9. REFERENCES

[1] B. Cabral and P. Marques. “Exception Handling: A Field Study in Java and .NET”. *ECOOP 2007 – Object-Oriented Programming,* pp. 151-175, 1 January 2007.

[2] B. Klatt, Z. Durdik, H. Koziolek, K. Krogmann, J. Stammel and R. Weiss. “Identify Impacts of Evolving Third Party Components on Long-living Software Systems”. *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on,* pp. 461-464, 2012.

[3] B.-M. Chang, J.-W. Jo and H. S. Her. “Visualization of Exception Propagation for Java using Static Analysis”.

*Proceedings Second IEEE Internation Workshop on Source Code Analysis and Manipulation,* pp. 173-182, 2002.

[4] C. Fan, M. Muller and I. Rezucha. "Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers". *Journal of the American Statistical Association,* pp. 387-402, 1962.

[5] C. Fu and B. G. Ryder. "Exception-chain Analysis: Revealing Exception Handling Architecture in Java Server Applications". *29th International Conference on Software Engineering (ICSE'07),* pp. 230-239, 2007.

[6] D. A. Thomas. "The Deplorable State of Class Libraries". *Journal of Object Technology,* pp. 21-27, 2002.

[7] D. S. Sena, R. Coelho, U. Kulesza and R. Bonifacio. "Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study". 2016. [Online]. Available: https://github.com/demost/pleamsr2016/wiki.

[8] E. A. Barbosa, A. Garcia and M. Mezini. "Heuristic Strategies for Recommendation of Exception Handling Code". *26th Brazilian Symposium on Software Engineering,* pp. 171-180, 2012.

[9] F. Ebert, F. Castor and A. Serebrenik. "An Exploratory Study on Exception Handling Bugs in Java Programs". *The Journal of Systems and Software,* pp. 82-101, 2015.

[10] G. Pinto, W. Torres, B. Fernandes, F. Castor and M. B. Roberto S. "A Large-Scale Study on The Usage of Java's Concurrent Programming Constructs". *Journal of Systems and Software,* pp. 59-81, 2015.

[11] H. Melo, R. Coelho, U. Kulesza and D. Sena. "In-depth Characterization of Exception Flows in Software Product Lines: An Empirical Study". *Journal of Software Engineering Research and Development,* 2013.

[12] H. Shah, C. Gorg and M. J. Harrold. "Visualization of Exception Handling Constructs to Support Program Understanding". *Proceedings of the 4th ACM Symposium on Software Visualization,* pp. 19-28, 2008.

[13] J. Bloch. "Effective Java". Person Education India, 2008.

[14] J. Gosling. "The Java Language Specification". Addison-Wesley Professional, 2000.

[15] M. Kechagia and S. Diomidis. "Undocumented and Unchecked: Exceptions That Spell Trouble". *Proceedings of the 11th Working Conference on Mining Software Repositories,* 2014.

[16] M. M. Rahman and C. K. Roy. "On the Use of Context in Recommending Exception Handling Code Examples". *IEEE 14th International Working Conference on Source Code Analysis and Manipulation,* pp. 285-294, 2014.

[17] M. P. Robillard and G. C. Murphy. "Designing Robust Java Programs with Exceptions". *International Conference on the Foundations of Software Engineering (FSE),* pp. 2-10, 2000.

[18] O. Shivers. "Control-Flow Analysis of High-Order Languages". PhD thesis, Carnegie-Mellon University, 1991.

[19] P. Clements and L. M. Northrop. "Software Product Lines: Practices and Patterns". Addison Wesley, 2001.

[20] P. Sawadpong, B. E. Allen and J. B. Willians. "Exception Handling Defects: An Empirical Study". *14th International Symposium on High-Assurance Systems Engineering,* pp. 90-97, 2012.

[21] R. Binder. "Testing Object-Oriented Systems: Models, Patterns and Tools". Addison-Wesley Professional, 2000.

[22] R. Coelho, A. G. A. Rashid, F. Ferrari, N. Cacho, U. Kulesza, A. Staa and C. Lucena. "Assessing the Impact of Aspects on Exception Flows: An Exploratory Study". *Proceedings of the 22nd European Conference on Object-Oriented,* pp. 207-234, 2008.

[23] R. J. Wirfs-Brock. "Towards Exception-Handling Best Practices and Patterns". *IEEE Software, V 23 (5), IEEE Computer Society,* pp. 11-13, 2006.

[24] R. Miller and A. Tripathi. "Issues with Exception Handling in Object-Oriented Systems". *ECOOP'97 - Object-Oriented Programming,* pp. 85-103, 1997.

[25] S. Gulwani and G. C. Necula. "Precise Interprocedural Analysis using Random Interpretation". *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pp. 324-337, 2005.

[26] S. Raemaekers, A. v. Deursen and J. Visser. "The Maven Repository Dataset of Metrics, Changes, and Dependencies". *Proceedings of the 10th Working Conference on Mining Software Repositories,* pp. 221-224, 2013.

[27] S. Raemaekers, A. van Deursen and J. Visser. "Exploring Risks in the Usage of Third-Party Libraries". *Software Improvement Group, Tech. Rep,* 2011.

[28] S. Sinha, A. Orso and M. J. Harrold. "Automated Support for Development, Maintenance and Testing in the Presence of Implicit Control Flow". *Proceedings in 26th International Conference on Software Engineering,* pp. 336-345, 23 May 2004.

[29] S. Thummalapenta and T. Xie. "Mining Exception-handling Rules As Sequence Association Rules". *Proceedings of the 31st International Conference on Software Engineering,* pp. 496-506, 2009.

[30] T. McCune. "Exception Handling Antipatterns". 04 06 2006. [Online]. Available: http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html.

[31] T. Yamane. "Statistics: An Introduction Analysis". New York, NY: Harper and Row, 1973.

[32] V. Bauer and L. Heinemann. "Understanding API Usage to Support Informed Decision Making". *Software Maintenance and Reengineering (CSMR), 2012 16th,* pp. 435-440, 2012.

[33] WALA, T. J.. "Watson Libraries for Analysis". Janeiro 2016. [Online]. Available: http://wala.sourceforge.net/.