On the Evolution of Exception Usage in Java Projects

Haidar Osman[+], Andrei Chiş[*], Jakob Schaerer[+], Mohammad Ghafari[+], and Oscar Nierstrasz[+]
[+]Software Composition Group, University of Bern, Switzerland
[*]Feenk GmbH, Switzerland

***Abstract*—Programming languages use exceptions to handle abnormal situations during the execution of a program. While programming languages often provide a set of standard exceptions, developers can further create custom exceptions to capture relevant data about project- and domain-specific errors. We hypothesize that, given their usefulness, custom exceptions are used increasingly as software systems mature. To assess this claim, we empirically analyze the evolution of exceptions and exception-handling code within four, popular and long-lived Java systems. We observe that indeed the amount of error-handling code, together with the number of custom exceptions and their usage in *catch* handlers and *throw* statements increase as projects evolve. However, we find that the usage of standard exceptions increases more than the usage of custom exceptions in both *catch* handlers and *throw* statements. A preliminary manual analysis of *throw* statements reveals that developers encode the domain information into the standard Java exceptions as custom string error messages instead of relying on custom exception classes.**

I. Introduction

Exception handling is a mechanism that allows developers to deal with abnormal execution flows of a program (*e.g.*, due to network failures, corrupted data). Given the pervasiveness of exceptions in today's programming languages, many studies have looked at how developers use exceptions in practice [1][2][3][4][5] and revealed that there is still a wide misuse of exceptions as an error-recovering mechanism. For example, developers use many empty *catch* handlers [2], catch and throw standard exceptions instead of specialized ones [2], or just ignore exception handling until an error occurs [4].

Custom exceptions can provide better error-handling support; they often provide domain specific information that eases the error handling and recovery. Given that relying on custom exceptions requires that developers hold enough knowledge about the project and its domain, we hypothesize that *at the beginning of a project, developers do not have a deep understanding of the project and its domain, so they rely on standard exceptions. As the system evolves and developers become more knowledgable about it, the usage of custom exceptions increases*.

To understand the various usages of exceptions in software systems and assess our hypothesis, we carry out an empirical investigation into the evolution of exceptions and exception-handling code within four long-lived Java projects. The selected projects are from different fields, have a wide user base, and are at least 7 years old. We formulate the following research questions and use them to guide our investigation:

RQ1. How does the amount of exception-handling code change as software systems evolve?

To answer this question, we extract all *catch* handlers from various versions of the analyzed systems. For each version, we compute the total number of lines of code (LOC) of *catch* handlers and compare its evolution with the overall LOC of the project. As expected, the LOC for exception-handling code increases along the evolution of a project, but the ratio of exception-handling code shows only slight variations.

RQ2. How does the usage of exceptions change as software systems evolve?

We mine multiple versions of each project and extract all defined exceptions and *throw* statements. We classify the exceptions into standard, custom, and third-party then analyze how the usage of these three categories evolves in the code. We observe that both the ratios of *catch* handlers and *throw* statements remain constant during the evolution of the projects.

RQ3. How are customized exceptions used during the evolution of software systems?

We realize that although developers define more custom exceptions and use them in *throw* and *catch* statements, they still rely more on standard exceptions with customized string error messages. We believe a further study is required to explore this phenomenon.

Our results reveal that exception usage patterns do not change during the evolution of software projects. The ratios of exception handling code and the usage of different types of exceptions remain generally constant. Also, we observe that developers do indeed encode domain-specific information together with thrown exceptions, however they add custom string messages to standard Java exceptions instead of relying on dedicated exception classes. This analysis indicates that developers could benefit from tool support, like dedicated refactoring, to extract custom exceptions from multiple *throw* statements throwing standard exceptions with similar or identical error messages. Mining exception usage is a known topic and various studies investigate how exceptions are used, particularly in Java. However, to the best of our knowledge, this is the first study on how exception usage evolves in long-lived systems. In the following sections, we explain our study design, and thoroughly discuss our findings that shed the light on further research in this domain.

II. Empirical Study

In this section we describe the design of our study and discuss its main findings. We implement a tool that carries out the cloning, analysis, and storage of the results. The tool is publicly accessible.[1]

[1]https://github.com/haidaros/evolution-of-exceptions

978-1-5090-5501-2/17/$31.00 © 2017 IEEE

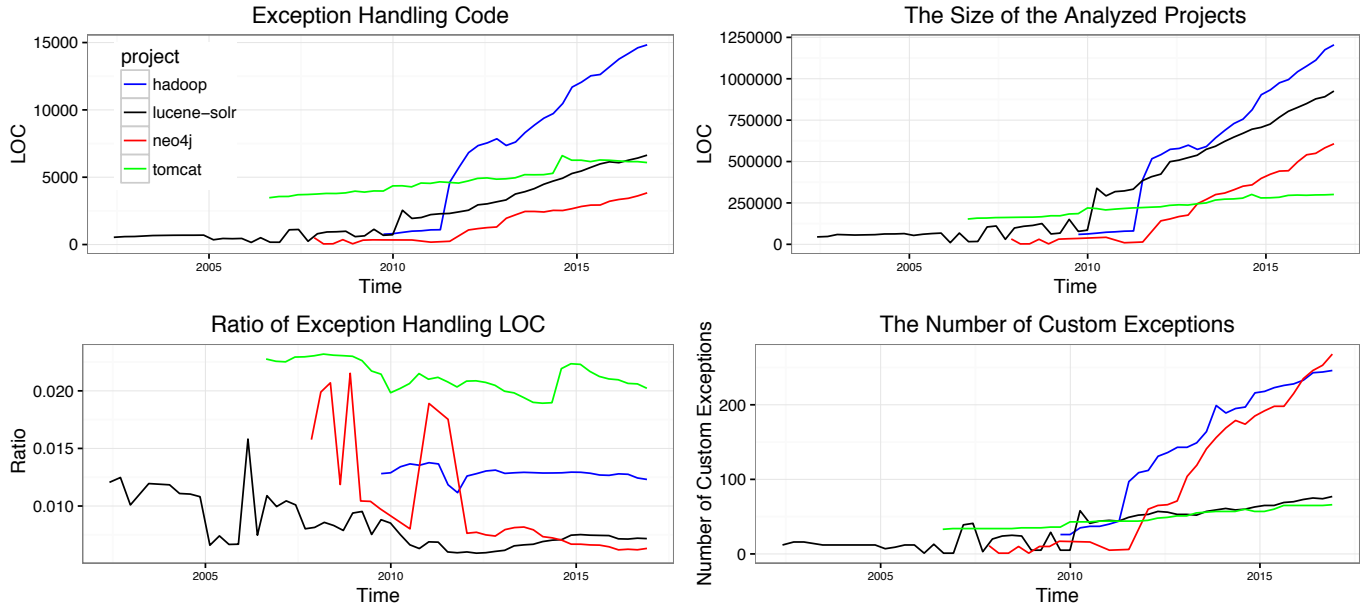

Fig. 1. The evolution of the size of exception handling code relatively to the evolution of the code base of the four studied projects.

A. Subjects

Previous work shows that different categories of software systems deal with exceptions differently [2]. Hence, for this study, we select four open-source Java systems from four different domains: Lucene-Solr[2] (search engine), Tomcat[3] (application server), Neo4j[4] (database system), and Hadoop[5] (big data infrastructure). Apache Solr is an open-source search engine built on top of Lucene,[6] the de facto library in Java for text indexing and searching. The two systems were merged into the Lucene-Solr project in 2010. Tomcat is an application server that can host web applications and services. Neo4j is a No-SQL graph database system. Hadoop is a framework for the development of scalable distributed data processing applications. Table I gives more details about these systems.

All chosen systems are mature, production-ready, and used heavily in industry. With more than 7 years of development, these long-living systems provide a good set-up for reasoning about the evolution of exception usage.

TABLE I
DETAILS ABOUT THE ANALYZED SYSTEMS, AS TO THE SUBMISSION DATE (01.12.2016)

System	Latest Release	#Months of Development	Java LOC
Lucene-Solr	6.3.0	182	$\approx 920K$
Tomcat	8.5.8	127	$\approx 300K$
Neo4j	3.0	114	$\approx 600K$
Hadoop	2.7.3	90	$\approx 1.2M$

[2]http://lucene.apache.org/solr
[3]http://tomcat.apache.org
[4]https://neo4j.com
[5]http://hadoop.apache.org
[6]http://lucene.apache.org/core

B. Definitions and Computed Metrics

In this study, a Java standard exception is an exception that is included in the Java standard libraries.[7] A custom exception is a Java exception, the source code of which is present in the project's repository. A third-party or a library exception is any exception that is neither custom nor standard.

To perform our study we first take a snapshot of each project's repository at three-month intervals. Then, for each snapshot, we extract the following metrics:

- The number of custom exceptions.
- The number and the ratio of catch handlers with a custom exception as a parameter.
- The number and the ratio of catch handlers with a standard Java exception as a parameter.
- The number and the ratio of catch handlers with a third-party (library) exception as a parameter.
- The number and the ratio of throw statements with a custom exception as a parameter.
- The number and the ratio of throw statements with a standard Java exception as a parameter.
- The number and the ratio of throw statements with a third-party (library) exception as a parameter.
- The number and the ratio of throw statements with a standard Java exception initialized using a string value.
- LOC_{catch}: The LOC of all catch blocks.
- LOC: The size of the system in LOC.

C. Results

The systems in our study are long-lived projects with active communities, as indicated by the evolution of LOC in Figure 1. Except for Tomcat, all other systems grew rapidly in size. The

[7]https://docs.oracle.com/javase/8/docs/api/overview-tree.html

amount of Java code in Solr, Hadoop, and Neo4j has more than doubled in the past three years. Tomcat also grew in size, but not as fast as the other systems.

To understand the evolution of error handling code (*RQ1*), we study the evolution of LOC_{catch} relative to the overall size of the studied systems ($LOC_{catch} \div LOC$) and in absolute values. As can be seen in Figure 1, not only the amount of exception handling code increases as the studied systems evolve, but also the number of defined custom exceptions. However, the exception handling code ratio remains generally constant in Hadoop, Solr, and Tomcat and even decreases in Neo4j. The ratio of exception handling code is between 0.02% and 2.5%. This indicates that the overall code base grows at a similar or faster pace than the exception handling code does.

To understand the evolution of exception usage (*RQ2*), we study the evolution of *catch* handlers and *throw* statements. We use, as a proxy for exception usage, the numbers and ratios of *catch* handlers and *throw* statements with different types of exceptions. We expect that as a system evolves, developers depend more often on custom exceptions in *catch* handlers and *throw* statements and less often on standard exceptions. Surprisingly, Figure 2 shows that the ratios of thrown and caught custom exceptions remain constant. Developers throw and catch standard Java exceptions more than custom exceptions throughout the evolution of the studied systems. Figure 2 also shows that the number of catch blocks with *java.lang.Exception* as a parameter, which is considered improper [1], does not decrease.

Intuitively, as a project evolves, its developers become more experienced with its behaviour. However, although the number of defined custom exceptions and their occurrences in *catch* handlers and *throw* statements increase, the usage pattern does not change. Developers still use standard exceptions more often even as they become, theoretically, more experienced.

This low usage of custom exceptions is interesting. We investigate deeper into the used standard exceptions and notice that most of the throw statements that have standard exceptions as parameters, actually contain a *String* parameter, as shown in Figure 2. A preliminary analysis of the last version of Solr shows that out of 3'796 statements that throw exceptions with *String* arguments, 429 appear identically in the code more than once. Furthermore, 40 of these statements appear identically more than five times. For instance, all the following throw statements appear in the code base more than 10 times:

```
1- throw new RuntimeException("Unable to load default
     stopword set");
2- throw new IOException("Fake IOException");
3- throw new IllegalArgumentException("this suggester
     doesn't support contexts");
4- throw new SQLException("Cannot be called from
     PreparedStatement");
```

These thrown standard exceptions encode project-specific errors in their *String* parameters. We also find some evidence that developers actually use these error messages to guide the error handling. For instance, the following code snippet from Solr uses the string message "*Fake IOException*" (the second example from the previous listing) to handle the error:

```
try {
    iw.close();
} catch (Exception e) {
    if (e.getMessage() != null && e.getMessage().
     startsWith("Fake IOException")) {
        exceptionStream.println("\nTEST: got expected
     fake exc:" + e.getMessage());
        e.printStackTrace(exceptionStream);
        try {
            iw.rollback();
        } catch (Throwable t) {
        }
    } else {
        Rethrow.rethrow(e);
    }
}
```

These exceptions can be refactored into custom exceptions to make the code more readable and maintainable. Thus, as an answer to the question of how developers customize exceptions (*RQ3*), developers not only define custom exceptions, but also rely heavily on standard exceptions with customized error messages.

To summarize, our results show that exception handling code grows at the same pace as the code base itself. Exception usage patterns do not appear to change during the lifetime of the studied systems, despite the fact that more custom exceptions are introduced as these systems evolve. Developers rely on standard exceptions with error messages that embed project-specific knowledge more than they rely on custom exceptions, a phenomenon that is a subject for further investigation.

III. Threats to Validity

We implemented our analysis in Java using JavaParser[8] to extract *throw* statements, *catch* handlers, and custom exception classes. To mitigate bugs in this analysis we created a second independent implementation using a different language and parser (*i.e.*, Pharo[9] and SmaCC[10]). We iterated over these implementations until their results matched. These implementations have different authors and use two mature parsers of different types. This increases our confidence in the correctness of the results. To compute LOC we used *cloc*.[11] Furthermore, as the analyzed projects are all open-source, we do not know if the same characteristics would be observed on closed-source projects.

For this analysis we only test our hypothesis on four open-source Java systems. The choice of systems directly influences the results. Selecting different types of systems can lead to different results. For example, frameworks can define more custom exceptions that become third-party exceptions in applications using those frameworks. Hence, we cannot make any claims regarding the generalizability of our findings. Due to the limitations of static analysis, we were unable to determine the exception being thrown for 46'619 throw statements out of 452'723 in all projects in all revisions. On average, 10% of the analyzed throw statements per project, and per revision were not categorized. This error rate does not have a significant influence on our results.

[8]https://github.com/javaparser/javaparser
[9]http://pharo.org
[10]http://www.smalltalkhub.com/#!/~JohnBrant/SmaCC
[11]https://github.com/AlDanial/cloc

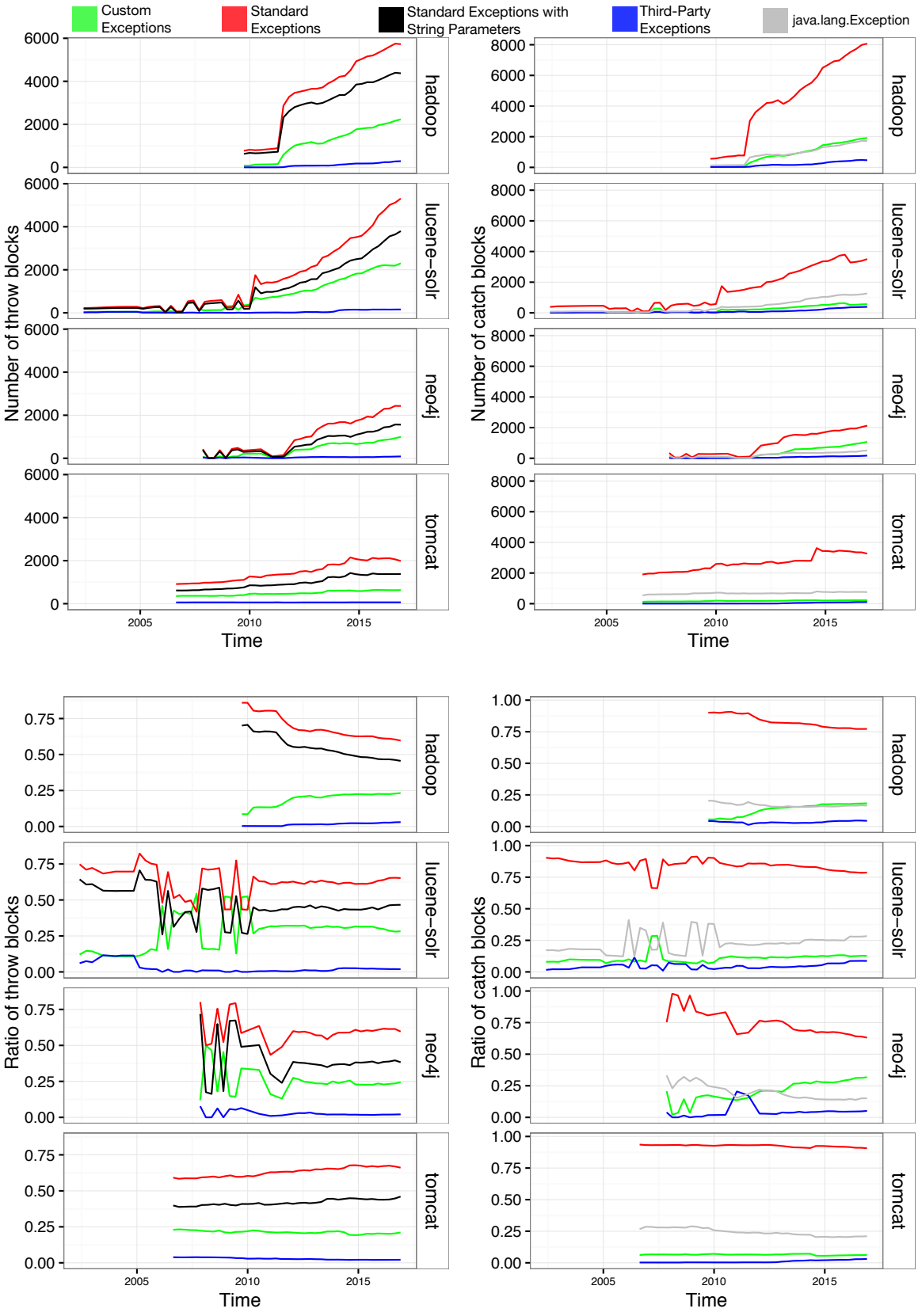

Fig. 2. The evolution of the throw statements and catch handlers with different types of exceptions. We can see that the ratios of used exceptions remain constant for all the different types. Clearly, standard exceptions are the most used throughout the evolution of the four projects.

IV. Related Work

We discuss related work that has studied usage of exceptions.

Asaduzzaman *et al.* [1] analyzed exception usage in 274K open source Java projects from GitHub. They found out that developers define their own exceptions to be mostly checked, revealing their intent to handle their exceptions and recover from them. Another important finding is that improper exception handling practices are as common with expert programmers as with novice programmers. This means that exception handling might be more dependent on the maturity of a project than on the expertise of programmers. In this study we do not take into account developer experience, however, we observe that the use of custom exceptions does not increase as systems evolve.

Cabral and Marques [2] analyzed exception handling in 32 applications in Java and .Net. They concluded that *"exceptions are not correctly used as an error recovery mechanism"*. They found that only 5% of code in Java systems is dedicated to error recovery. Developers mostly either just log exceptions or do nothing (empty catch block). In our study, we observe that error handling code is only between 0.02% and 2.5% of the Java code. The difference between the two studies is due to the fact that the two analyzed datasets contain different systems. Cabral and Marques also revealed that catching *java.lang.Exception* is common, although they considered it to be bad exception usage. We observe that the ratio of *catch* blocks handling *java.lang.Exception* has only minor variations.

Nakshatri *et al.* [3] compared exception usage in practice with exception usage best practices. They concluded that most developers do not follow the rules. Top-level exceptions (*e.g.*, *Exception, IOException*) are caught more often than specific ones. Exceptions are mostly left unattended or at best logged.

Shah *et al.* [4] revealed some possible reasons behind the mismatch between theoretical and practical exception usage. The participants in the study stated that they use exceptions primarily for debugging and they adopt an *ignore-for-now* strategy when handling exceptions. They ignore exception handling until an error occurs. The participants think of exception handling as a "waste of time" and they expressed their appreciation for languages that do not force them to deal with exceptions. However, one participant, who is significantly more experienced, had a different attitude and expressed an appreciation of checked exceptions in Java.

Sena *et al.* [5] employed exception flow analysis and manual inspection to understand exception handling strategies in Java libraries. As in previous studies, the authors also found many anti-patterns. However, Sena *et al.* went further and discovered that more than 20% of reported bugs in the 7 most popular Java libraries are related to improper exception usage (*e.g.*, as catch-and-ignore or catch *java.lang.Exception*). To address this problem, many studies recommend exception handling code based on mined knowledge from software repositories [6][7][8].

This work complements these studies with a new perspective by looking at how exception usage evolves. To our knowledge this is the first study analyzing the usage of different types of exceptions in *catch* handlers and *throw* statements.

V. Conclusions and Future Work

The promise of exception handling is that, if correctly used, it enables systems to gracefully recover from abnormal situations, without having to pollute the main logic of a program with checks for special error codes. Nonetheless, previous research shows that developers do not take advantage of custom exceptions. To better understand how exception usage changes as software systems evolve we analyze the evolution of exceptions and exception-handling code within four, popular, and long-lived Java systems. We observe that the ratios of custom exception, *catch* blocks and *throw* statements remain constant even when systems quadrupled in size. We further notice that, throughout the evolution of a system, developers prefer to encode error-relevant information through plain strings and standard exceptions instead of creating custom ones.

This work provides only an initial insight into the evolution of exceptions. A future work track consists in increasing the number of analyzed projects and looking for different patterns of exception evolution. Also we plan to investigate whether enough domain specific knowledge is encoded in these custom messages and why developers adopt such an ad-hoc way of exception reporting rather than throwing custom exceptions. Finally, another future work would be using these findings to provide developers with better tooling for discovering and transforming sets of *throw* statements of standard exceptions with similar error messages into custom exceptions.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, 01.01.2016 to 30.12.2018). We also acknowledge the financial support of the Swiss Object-Oriented Systems and Environments (CHOOSE)[12] for the presentation of this research.

References

[1] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in Java?" in *MSR '16*. New York, NY, USA: ACM, 2016, pp. 516–519.

[2] B. Cabral and P. Marques, "Exception handling: A field study in Java and .NET," in *Proceedings of European Conference on Object-Oriented Programming*, ser. LNCS, vol. 4609. Springer Verlag, 2007, pp. 151–175.

[3] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: An empirical study," in *MSR '16*. New York, NY, USA: ACM, 2016, pp. 500–503.

[4] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in *Proceedings of the 4th international workshop on Exception handling*. ACM, 2008, pp. 62–68.

[5] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of Java libraries: An empirical study," in *MSR '16*. New York, NY, USA: ACM, 2016, pp. 212–222.

[6] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples." in *SCAM '14*, 2014, pp. 285–294.

[7] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *Software Engineering (SBES), 2012 26th Brazilian Symposium on*. IEEE, 2012, pp. 171–180.

[8] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 496–506.

[12] http://www.choose.s-i.ch