Analysis of Exception Handling Patterns in Java Projects: An Empirical Study

Suman Nakshatri, Maithri Hegde, Sahithi Thandra
David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada
{sdnaksha, m2hegde, sthandra}@uwaterloo.ca

ABSTRACT

Exception handling is a powerful tool provided by many programming languages to help developers deal with unforeseen conditions. Java is one of the few programming languages to enforce an additional compilation check on certain subclasses of the Exception class through *checked exceptions*. As part of this study, empirical data was extracted from software projects developed in Java. The intent is to explore how developers respond to checked exceptions and identify common patterns used by them to deal with exceptions, checked or otherwise. Bloch's book - *"Effective Java"* [1] was used as reference for best practices in exception handling - these recommendations were compared against results from the empirical data. Results of this study indicate that most programmers ignore checked exceptions and leave them unnoticed. Additionally, it is observed that classes higher in the exception class hierarchy are more frequently used as compared to specific exception subclasses.

CCS Concepts

•Software and its engineering → Error handling and recovery;

Keywords

Java Exception Handling; Github; Best Practices; Boa

1. INTRODUCTION

Java provides a powerful support system for exceptions involving dedicated try-catch-finally blocks to separate functional code from exception-handling logic. *Throwable* is the superclass of all exceptions and errors (which are unhandled critical problems). In this paper, the focus is exclusively on exceptions. The terms error(s) and exception(s) are used interchangeably, however in all cases they refer to exceptions and not the category of serious errors, unless specified otherwise.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR'16, May 14-15, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2901739.2903499

Table 1: Dataset Statistics for GitHub

Total projects	7,830,023
Number of Java projects	554,864
Methods with atleast one catch block	10,862,172
Total number of Catch blocks	16,172,462

Table 2: Dataset Statistics for SourceForge

Total projects	699,331
Number of Java projects	50,692
Methods with atleast one catch block	6,657,595
Total number of Catch blocks	9,956,760

One noted advantage of exception handling in Java is its hierarchy and classification of exception types based on the type of error. The highest in this hierarchy is the class *Exception*. All subclasses of *Exception* which do not fall under RuntimeException are categorized as checked exceptions. The term *checked* has come into usage as Java enforces or *checks* for these exceptions during compilation. The intention is to force developers to think of sophisticated ways to handle such exceptions in case they occur. In this paper, we mine data to gain insight into how developers handle checked exceptions and the utility of Java enforcing the same. An extension to this study includes extracting general exception handling patterns used by developers.

The specificity of an error caught increases as we move down the class hierarchy. To illustrate, the exception SQLClientInfoException inherits properties of its parent class SQLException. Catching an SQLClientInfoException would provide extra information of the error by fetching details regarding SQL client properties, going beyond what is provided by superclass SQLException. Furthermore, since each exception is an object, an exception thrown at a lower level can also be caught by its superclass. In this paper, we explore the frequency and usage of generic/ top-level exceptions. The research questions that guide our work are:

1. How well are checked exceptions handled? Does Java enforcing them fulfill its purpose?
2. What scenarios are exception handling used for?
3. How often are top-level exceptions caught and their potential danger(s) to the product quality?

Table 3: Top Exceptions Caught in Catch Blocks

ExceptionClass	Number	ExceptionType
Exception	3,859,217	Superclass
IOException	2,170,519	Checked
SQLException	681,638	Checked
Throwable	670,214	Superclass
InterruptedException	555,555	Checked
IllegalArgumentException	460,872	Unchecked
NumberFormatException	364,362	Unchecked
NullPointerException	326,193	Unchecked
RemoteException	263,920	Checked
RecognitionException	203,892	Unchecked
TOTAL Checked	3,671,632	
TOTAL UnChecked	1,355,319	

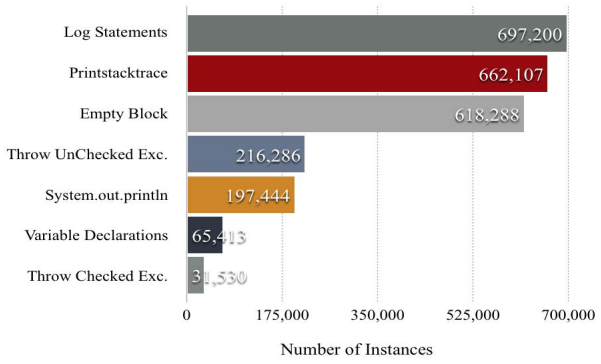

Figure 1: Top Operations performed in Checked Exception catch blocks

2. RELATED WORK

Exception handling behavior is a topic of long term study by software engineering researchers and product managers. Shah et al.[5] studied the developer perspective on exception handling and attempted to understand the root cause for developer attitude and actions. The basis of our study is to support or negate these qualitative claims with empirical data. Monperrus et al.[3] compared best practices fetched from known and authentic sources against empirical data extracted from Java projects. While this was an empirical study, the dataset used was small (32 Java projects). Moreover, the focus was not to specify patterns used in exception handling, but to challenge or validate known best practices. The contribution of our study, on the other hand, is in producing empirical data fetched from a large scale data source, with the intent of identifying common practices followed by developers to tackle exceptions.

3. RESEARCH METHODOLOGY

Analysis data was extracted from MSR'16 Mining Challenge Boa Dataset[2] using the BOA domain specific language[2] run on their web interface[4]. This dataset contains metadata for almost 700,000 SourceForge and 8,000,000 GitHub repositories. The latest snapshot was picked up for the purpose of this study and a total of 554,864 Java projects in the GitHub repository were analyzed. The dataset is current as of September 2015. A study on 50,692 Java projects from the SourceForge Sep' 13 dataset was also performed to validate GitHub results, however GitHub being much larger and more current, all results and figures in this paper correspond only to data extracted from it. Table 1 and 2 give brief statistics on both datasets studied.

3.1 Mining Exception Categories

The number of exceptions handled in a method can be extracted using one or any combination of the following three methods:

1. Exceptions thrown using *throws* keyword in the method signature.

2. Exceptions thrown using *throw* keyword in the method body.

3. The exceptions caught in a *try-catch* block of a method.

The first method may not provide a true picture as exceptions thrown using *throws* in the method signature are falsely added with the call stack of methods propagating the exception until it is caught. Furthermore, an exception thrown using the second method will eventually be caught by a caller method using a try catch block. Hence, the third method was chosen for our analysis. Table 3 gives the top twelve exceptions caught in catch block. For each exception in the table, both long and short notations were considered - for eg., the long notation for *IOException* would be *java.io.IOException*.

4. RESULTS

The cumulative results in Table 3 show that checked exceptions account for almost three times the number of unchecked exceptions in Java projects. This can be attributed to the compile-time checking for the presence of checked exception handlers.

In the following sections, mined results on the usefulness of checked exceptions are presented. Further insights into: how programmers handle said exceptions; value of enforcement, and general exception handling patterns are provided.

It can be inferred from the above table that developers often catch exceptions at the 'top-level' using the *Throwable* and *Exception* class. Further details about generic exception handling and potential harmful effects to code quality are discussed in Section 4.3.

4.1 Analyzing Checked Exception Handling

As discussed in previous sections, Java enforces compile-time use of checked exceptions to compel one to envision possible errors that could occur and tackle them appropriately. To illustrate the use of SQLException handling, if a primary server crash is expected, the catch clause could be coded to connect to a secondary server or return an appropriate message back to client. This enforcement is designed to improve application developer experience and improve code quality.

Usage of the top three checked exceptions picked from Table 3 was analyzed. This began with mining the top twenty *method* calls; *throw an exception* calls and *variable declarations* made inside a catch construct of a checked exception, as shown in Fig. 1. It can be observed that the top spot is shared between *printStrackTrace* and log methods, which

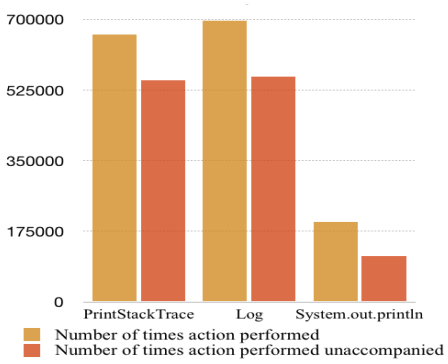

Figure 2: Comparison of top unaccompanied operations performed vs their total instances in catch blocks of Checked Exceptions.

```
try {
        while(rs.next()) {
                Item temp = new Item(rs.getInt("itemid"), rs.getString("title"));
                owned_items.add(temp);
        }
} catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}
```

Figure 3: Code Snippet from a GitHub project.

indicates that many developers use checked exceptions for logging and debugging. The second major observation is the large number of empty catch blocks - this is discussed in Section 4.2. However, the operations shown in Fig. 1 may have accompanied other operations inside the catch block. For example, a *log* call may have been made along with a *throw exception* call. For additional clarity, results for common unaccompanied operations performed inside a checked exception handling construct, as shown in Fig. 2, were mined. The phrase *"unaccompanied operation"* implies that operation in the figure was the only action taken inside the catch block. It can be noted that 83% of the *printStackTrace* operations were performed unaccompanied. Many developers use popular IDEs such as Eclipse and IntelliJ which provide powerful suggestion and code completion tools, including adding try catch constructs with default *printStackTrace* call for checked exceptions. Fig. 3 gives an example of a code snippet extracted from the GitHub project *"pennshare"*, which shows the default checked *SQLException* handling.The high number of *printStackTrace* calls reveals that a lot of developers ignore and leave the catch block unattended.

Furthermore, it is also noticed that developers tend to wrap checked exceptions inside unchecked ones while throwing them back to the caller. Since access to plain source code is not provided, we may not be able to judge the reason behind these wrapped throws. However, Bloch recommends in his book [1] to *"use checked exceptions for recoverable conditions and runtime exceptions for programming errors"*. It states that if the user finds a checked exception unrecoverable, wrapping into an unchecked exception keeping the stack trace intact and throwing it back to the client or the caller method is justifiable. The data indicates that around 74% of throw statements in IOException handlers and 65% of throw statements in SQLException handlers were made to RuntimeException, which is consistent with Bloch's views.

This leads us to answer our first research question. Though checked exceptions were introduced to direct the thought process towards error recovery, the results show that most developers neglect handling them. This defeats the purpose of the Java enforcement mechanism.

4.2 Analyzing General Exception Handling

The data related to checked exceptions noted in Section 4.1 seems to extrapolate well with all types of exceptions. In this section, we will focus more on general exception handling patterns.

The number of empty catch blocks in Fig. 4 draws attention. The percentage of empty blocks was calculated with respect to the total number of catch blocks present in the dataset. It was found that 20% of the total (16,172,462) catch blocks were left empty. As stated by Bloch [1]*"An empty catch block defeats the purpose of exceptions, which is to force you to handle exceptional conditions"*. This holds the product at risk for the following reasons: time consuming debugging due to absence of tracing and logging when an error occurs; and swallowing issues not intended to be caught resulting in false positive output[1].

The top method calls in catch constructs show significantly more number of logging techniques as compared to the ones found in previous section. These include: log4j/ slf4j methods at all five levels (info, debug, etc.); handler module methods such as ReportError; logger module methods at various levels (severe, warning, etc.). All these method calls made for logging were added to produce the final number shown in Fig. 4 for 'Log Statements'.

Fig. 4 and Fig. 5 show a high usage of logging inside catch constructs; in particular, of log messages being used as standalone actions. This supports the claim in paper [5] - *"developers have shifted their perspective on exception handling from the intended proactive approach (i.e., how to handle possible exceptions) to a reactive approach (i.e., using exception handling as debugging aids)"*. The *printStackTrace* number also backs this claim.

Bloch advises that *"higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction"*[1]. To study this scenario, top hundred throw calls were extracted in the form of *'A -> B'* where A/B are exception class types and A throws B in its handler. The highest number of translations (145,916) were made from Exception to RuntimeException which runs counter to Bloch's advice. Most conversions from any exception type were made to RuntimeException. Excluding the above number and looking at next top twenty results, 66% conversion from lower to higher level exception and 24% from higher to lower are observed. The remaining were thrown at the same level, for eg. 'IOException throws IOException'. These cases show positive alignment with the advice.

We see that many developers seem to use Java exception handling for logging and fetching the stack trace. Many more developers choose to leave the catch constructs empty. Quite a lot of them convert and throw back exceptions to their caller methods. Due to the size of the dataset, it was

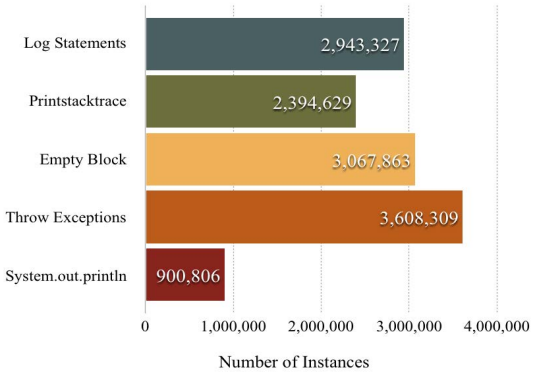

Figure 4: Top operations used in exception handling.

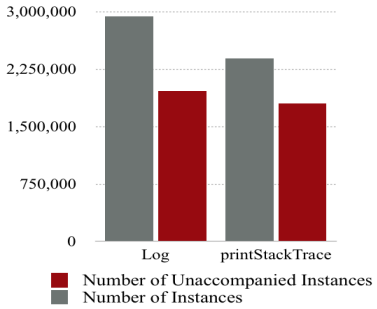

Figure 5: Comparison of top unaccompanied operations performed vs their total instances in exception handling.

challenging to fetch user-defined exceptions and meaningful operations performed as they are specific to each project. This answers our second research question.

4.3 Catching Top-level or Generic Exceptions

As showcased in Table 3, data of the top twenty exceptions caught shows that the superclass of all exceptions viz. *Exception* has the largest frequency of occurrence and that its base class *Throwable* also has a surprisingly high usage. As per findings in [5], many developers tend to keep exception handling as simple as possible, which potentially encourages wrapping of the calling code in a single *Exception* or *Throwable* class handler.

The numbers in Table 3 for *Exception* and *Throwable* includes code wrapped in multiple handlers along with generic handlers. To learn more, the number of methods that contained only *Exception* and *Throwable* handlers was analyzed further. A staggering 78% of the methods that caught *Exception* did not catch any of its subclasses and a similar observation of 84% was made for *Throwable*. A total of 10,862,172 methods with catch clauses were extracted. 30% of them contained only the generic *Exception* class handler.

Bloch [1] asserts - *"Exception handlers that are too general can make code more error-prone by catching and handling exceptions that were not anticipated by the programmer and for which the handler was not intended"*. Printing stack trace and logging were the top actions executed in an *Exception* catch construct. This makes the code more vulnerable as generic handlers provide low scope for specific log messages and recovery actions. *Throwable* handlers place a higher risk on the product as they also capture critical errors, which are not meant to be caught.

Finally, answering our third and last research question, *Exception* is the leading handler used, followed shortly by *Throwable*. Reiterating Bloch's views[1], this practice places a higher risk on the final product by catching unrelated bugs and discouraging specific meaningful recovery.

We performed a compare-and-contrast of GitHub results with data extracted from SourceForge projects. A similar trend for exception handling was observed in them.

5. THREATS TO VALIDITY

The material extracted from the data set provides empirical results on the exception types, number of exceptions and actions taken within a catch construct. However, due to lack of access to raw source code, the rationale behind these actions is unknown. Hence, our reasoning behind how and why handlers were used and categorizing them as incorrect may not always be sound. Since the intent of the study has been to align with best practices stated by Bloch[1], the focus here has been to surface patterns which stand for or against those views supported by empirical data.

6. CONCLUSION

In this paper, we studied all Java projects in the GitHub and SourceForge repositories to spot common exception handling practices. These were compared against best practices presented by Bloch[1]. To sum up, most developers ignore checked exceptions and leave them unattended. General exception handling shows developer inclination shifting from proactive recovery to debugging. Developers also tend to use generic handlers over specific ones. The reason for these approaches cannot be confirmed to be correct or incorrect without scanning source codes. However overall, the results show major differences between theoretical best practices and the use of exception handling in practice.

7. REFERENCES

[1] J. Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[2] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431, 2013.

[3] M. Monperrus, M. Germain De Montauzan, B. Cornu, R. Marvie, and R. Rouvoy. Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of 32 Java Software Packages. Technical report, Laboratoire d'Informatique Fondamentale de Lille, 2014.

[4] H. Rajan, T. N. Nguyen, R. Dyer, and H. A. Nguyen. Boa website. http://boa.cs.iastate.edu/, 2015.

[5] H. Shah, C. Görg, and M. J. Harrold. Why do developers neglect exception handling? In *Proceedings of the 4th International Workshop on Exception Handling*, WEH '08, pages 62–68, New York, NY, USA, 2008. ACM.