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

How Developers Use Exception Handling in Java?

Muhammad Asaduzzaman
University of Saskatchewan
md.asad@usask.ca

Muhammad Ahasanuzzaman
University of Dhaka
ahsan.du2010@gmail.com

Chanchal K. Roy
University of Saskatchewan
chanchal.roy@usask.ca

Kevin A. Schneider
University of Saskatchewan
kevin.schneider@usask.ca

ABSTRACT

Exception handling is a technique that addresses exceptional conditions in applications, allowing the normal flow of execution to continue in the event of an exception and/or to report on such events. Although exception handling techniques, features and bad coding practices have been discussed both in developer communities and in the literature, there is a marked lack of empirical evidence on how developers use exception handling in practice. In this paper we use the Boa language and infrastructure to analyze 274k open source Java projects in GitHub to discover how developers use exception handling. We not only consider various exception handling features but also explore bad coding practices and their relation to the experience of developers. Our results provide some interesting insights. For example, we found that bad exception handling coding practices are common in open source Java projects and regardless of experience all developers use bad exception handling coding practices.

CCS Concepts

•**Software and its engineering** → *Error handling and recovery;*

Keywords

Java; exception; language feature; source code mining

1. INTRODUCTION

An exception is an exceptional event that occurs during the execution of a program and can disrupt the normal flow of execution. To enable programmers to deal with such exceptional situations, modern programming languages have built-in support for exception handling. For example, the Java programming language uses several language constructs (such as try, catch, finally and throw) to support exception handling. Code that might throw exceptions needs to be enclosed in a **try** statement that catches those exceptions. The **try** statement is supported by **catch** and **finally** clauses that contain instructions specifying the actions needing to be taken when the exception occurs. Exception handling offers a number of advantages. This includes separating error handling code from the main logic, differentiating and grouping different exceptional situations and enabling programs to deal with errors.

It is required that developers follow the suggested guideline in the Java Language specifications [1] to enjoy the benefits of exception handling. While a number of techniques have been developed to identify causes of exceptions, suggesting exception handling code or to identify web discussions pertaining to exceptions, there is a marked lack of empirical evidence of how developers use exception handling in practice. In this paper we utilize the Boa language and infrastructure [1] to answer questions regarding how developers use exception handling in Java. These questions are selected to explore bad exception handling coding practices, their relationship to the experience of developers, using exception chaining, defining custom exception classes and using new exception handling features.

The remainder of the paper is organized as follows. Section 2 describes the data set used in our study. Section 3 briefly explains the exception handling technique in Java. Section 4 presents our research questions including the results of our empirical study. Section 5 summarizes the related work and Section 6 discusses threats to this study. Finally, Section 7 concludes the paper.

2. DATA SET

The data set used in this study is the 2015 GitHub data set from Boa. This includes all Java projects on GitHub with at least one or more Git repositories. The GitHub data set represents 274k projects with 22 million revisions by 320k developers. It consists of more than 120 million files and over 20 million of them are unique Java source files. Since we are interested in finding how developers are handling exceptions in Java without any constraints, we include small projects as well as large ones.

3. EXCEPTION HANDLING IN JAVA

This section briefly describes exception handling in Java. The language supports three different kinds of exceptions. These include checked exception, unchecked exception, and

[1]https://docs.oracle.com/javase/specs/

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

DOI: http://dx.doi.org/10.1145/2901739.2903500

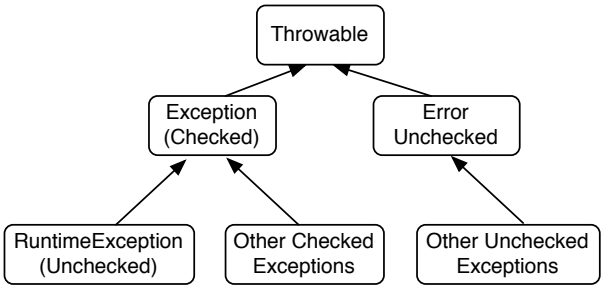

Figure 1: Exception Hierarchy in Java

error (see Fig. 1). Checked exceptions are those exceptions that a well written Java application should handle and recover from. Code that anticipates checked exceptions must follow the catch or specify requirements in Java. The requirements are as follows. The code should enclosed by a **try** statement and must be followed by exception handlers. If an exception occurs in the **try** block, the exception will be handled by the exception handlers. The **try** block must be followed by either one or more **catch** blocks, a **finally** block or a combination of both to handle exceptions. If a method throws one or more exceptions it must list those exceptions using a *throws* clause in its declaration. The second kind of exception is the error. The causes of these exceptions are external to the applications and the applications cannot recover from these exceptions. These are identified by **Error** and its subclasses. The last kind of exception are those exceptions that are internal to an application and that the application cannot recover from it. These exceptions are not subject to catch or specify requirements. They are indicated by the **Runtime** exception and its subclasses. Exceptions that are not indicated by **Error**, **RuntimeException** or their subclasses, are checked exceptions.

4. HOW DEVELOPERS USE EXCEPTION HANDLING IN JAVA

This section answers our research questions regarding exception handling in Java. In addition to discussing the incorrect use of exception handling we also investigate its relation to developer experience, when new exception handling features are used, patterns of use in exception chaining and also how developers create their own exception classes.

4.1 Exception Handling Coding Practices to Avoid

To identify improper exception handling coding practices we review Java Language Specifications, previous research papers [6, 4], software information sites, developer blogs and books [5]. We identify the following coding patterns that need to be avoided. While this may not be a complete list, they do represent the majority of the improper exception handling coding practices.

- **Ignoring exceptions (IE):** In this case developers leave the catch or finally block empty. This defeats the purpose of exceptions because this prevents programs from recovering from exceptions.

- **Catching unchecked exception (CUE)**: Unchecked exceptions are the result of programming errors that could have been avoided by checking the proper conditions. Generally, unchecked exceptions should not be caught, although there are exceptions to this practice.

- **Not preserving original exception** (NPOE): Instead of handling exceptions at the lower level, developers use the **throw** statement to throw exceptions to the higher level in response to another exception. If they forget to wrap the original exception object within the new exception object, they are throwing the exception but losing the original source of the problem.

- **Use generic exception handler (UGEH)**: Instead of catching specific exceptions developers use a single catch block to collect all exceptions. As a result it may be difficult to determine why the exception was thrown. consequently the runtime system cannot attempt recovery.

- **Catching Error (CE)**: Applications cannot recover from errors that are caused by the environment in which the application is running. All errors in Java are of type **java.lang.Error**. Thus, developers should not catch exceptions indicated by **Error** or its subclasses.

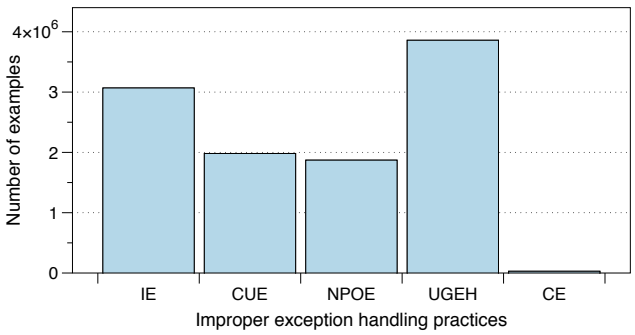

Figure 2: The frequency of different improper exception handling coding practices

We are interested in finding how frequent the improper exception handling coding practices are in source code repositories. Figure 2 shows the frequency of those coding practices in the GitHub data set. The figure shows that the data set contains a significant number of all five bad coding practices. We find that using an generic exception handler is the most frequent one. This is an indication that developers are very reluctant in using exception handling. Next is ignoring exceptions. Many developers are not aware of how to recover from exceptions or do not care to and thus leave the **catch** and **finally** blocks empty. This could cause difficulties when maintaining applications as well as leave potential bugs in the code. We found a very small number of code examples where developers catch an error, indicating that most developers are aware of this bad practice.

4.2 How developers use exception chaining

In many applications lower level methods are required to propagate information regarding exceptions to a higher level. This enables applications to notify end users about exceptions and users can take appropriate actions at the more abstract level. Exception chaining is the mechanism that

Table 1: Patterns of expressions used for exception chaining

Expression Type	Percent	Example
Cast	1.0215	**catch**(CustomException ex) { **throw** (UnsupportredEncodingException) ex; }
Conditional	0.0535	**catch**(Exception ex) { error_code==0 ? **throw new** RuntimeException("Error message",ex): **throw new** Exception("Another message",ex); }
Method Call	5.8479	**catch**(Expression ex) { **throw** InvocationTargetException.getCause(); }
New	76.1696	**catch**(Expression ex) { **throw new** Exception("Additional error message",ex); }
Variable Access	16.8920	**catch**(Expression ex) { **throw** ex; }
Assignment	0.0147	**catch**(Expression ex) { **throw** lastException = **new** KeyStrokeException(ex); }
Null	0.0008	**catch**(Expression ex) { **return** null; }

allows applications to propagate exceptions up the call stack and at the same time preserve important error information. To use exception chaining an application uses the *throw* statement to throw an exception in response to another exception. We investigated the patterns developers used to throw exceptions. This data can help other developers learn the various ways **throw** statements are constructed. Table 1 shows the different expression types developers used to throw exceptions in response to another exception. We see that the largest number of **throw** statements in exception chaining throws an exception with a new exception object that incorporates an additional error message and may also contain a reference to the lower level exception. Conditional expressions can be used to throw different expression objects with different error message information depending on some conditions. Assignment and null expressions are very infrequent. Variable access is the second most common expression type used and method call is the third most common expression type used in exception chaining.

4.3 How do developers define their own exceptions?

Developers can define their own exceptions by extending any subclasses of **Throwable**. Since unchecked exceptions (**RuntimeException**, **Error** and their subclasses) do not require to fulfill catch or specify requirements it may be the case that developers are tempted to create all exceptions by extending RuntimeException. The official documentation of Java suggests that if a client can expect to recover from an exception, make it a checked exception; otherwise make it an unchecked exception. We are interested to find out how developers create their own exceptions. The data can show us which options developers prefer to use in practice. Interestingly when we investigate this in the GitHub data set we found that in majority of the cases developers define their own exception classes by extending Exception. This indicates that when developers create their own exceptions, they are concerned about error recovery. We also observe evidence where developers create exception classes by extending **RuntimeExcepion**, **Throwable** or **Error**, but those cases are very few in number.

4.4 When do developers use new exception handling features?

Java SE 7 introduced a number of changes to the exception handling mechanism. These included catching multiple exceptions by using a single catch statement (also knows as multi-catch), a try-with-resources declaration that frees developers from explicitly closing resources, an AutoCloseable interface (classes that want to take advantage of try-with-resources need to implement the close method of the AutoCloseable interface). We are interested in finding whether developers use these new features in their code. Fig 3 shows how many files incorporated a new feature each month. From the figure we can see that not only do developers use the new features, but also developers start using these features long before their release (these features were officially released in July 2011).

4.5 Does developer experience affect exception handling coding practice?

We are interested in finding out how developer experience affects bad exception handling coding practices. To calculate developer experience, we use the total number of revisions committed to the source code by a developer up to a particular point in time. In our analysis, we use September 2015 as the particular point in time. We calculate the experience using the weighted mean of the number of committed revisions of all contributing developers as per Mockus and Weiss [7]. The weight is the proportion of the commit of a particular project and experience is their general experience at that particular time. Here, we categorize developers in five categories using their experience calculated by the above process and they are: 1) Novice; 2) Beginner; 3) Competent; 4) Proficient; and, 5) Expert. Then, for each bad exception handling coding practice we determine the ratio of bad code examples to total examples for all five developer experience groups. Results from our study (see Fig. 4) shows that regardless of the experience all developers exhibit improper exception handling coding practices. However, for IE, CUE and UGEH improper exception handling categories, novice developers contribute the most.

5. RELATED WORK

A number of studies have been conducted on exception handling. Weimer and Necula [6] used data flow analysis to find and characterize exception handling mistakes in resource management. Cabral and Marques [8] analyzed 32 Java and .NET applications and found that exception handling is not used as an error handling tool; when exceptions occur applications take different recovery actions. Thummalapenta and Xie [3] developed a technique that mines exception handling rules as a sequence of association rules. In another study [4] they reported that exception handling actions can be conditional and may need to accommodate exceptional cases. However, none of these studies explore improper exception handling coding practices or when new exception handling features are introduced in the code. The

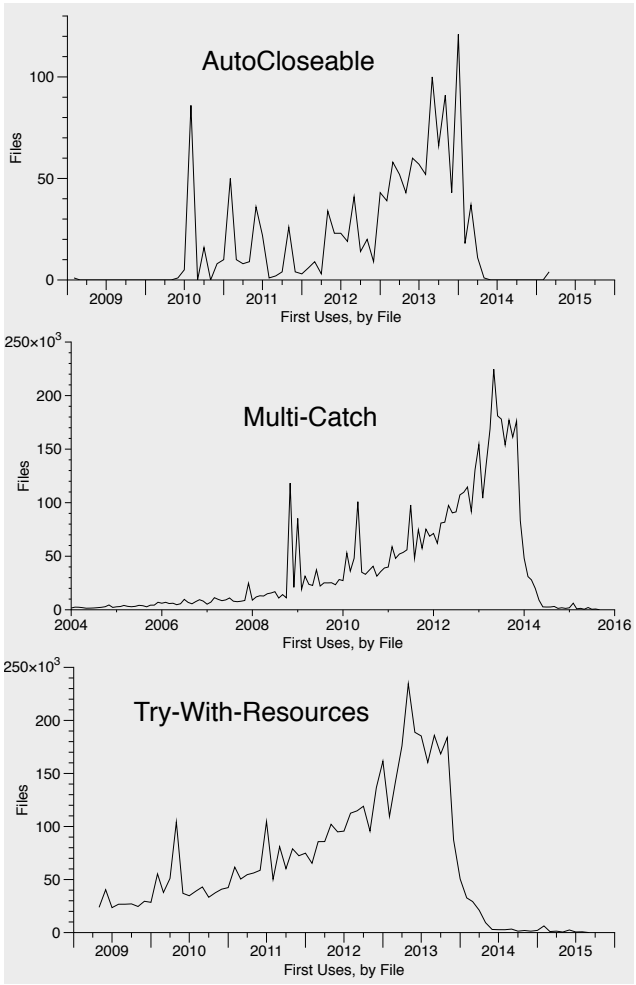

Figure 3: How many files incorporated a new exception handling feature each month

most relevant study to ours is the work of Dyer et al. [2] that used the Boa infrastructure to analyze the use of Java language features over time. While their work was to determine how new features are adopted once released in Java, we focused our attention on how developers use exception handling in Java.

6. THREATS TO VALIDITY

There are a number of threats to this study. First, we created programs in the Boa language to collect answers to our research questions. Although we cannot guarantee that our implementation does not contain errors, we spent a considerable amount of time testing to minimize the possibility of introducing errors. Second, in this study we investigated open source Java projects on GitHub to determine whether developers exhibit improper exception handling coding practices or not. This could be due to the limitation of the programming language or application requirements. However, we did not investigate the reasons behind improper use of exception handling in this study. Third, we only studied open source Java projects in this study. Therefore, we are unable to generalize our result to non-open source Java projects.

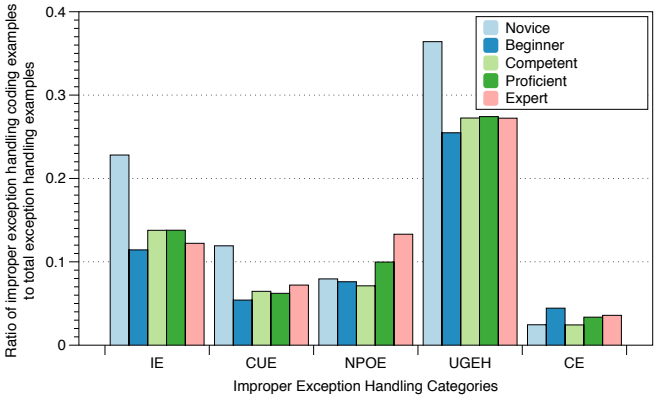

Figure 4: The ratio of different improper exception handling coding examples to total examples for different developer experience groups

7. CONCLUSION

In this paper we investigated exception handling in Java using the ultra large scale data set (274k Java projects) provided by the Boa infrastructure. We use the Boa language and infrastructure for both creating queries and collecting answers to our research questions. Our study reveals that improper exception handling coding practice is not uncommon in Java applications and regardless of experience all developers use them. Through our investigation on how developers use exception chaining and how they define their own exceptions, we found that developers typically follow the official Java language guidelines. We also investigated the new exception handling features of Java and found that developers use new exception handling features prior to their official support in a Java release. The programs and additional study results can be found online[2].

8. REFERENCES

[1] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories", in Proc. of ICSE, 2013, pp. 422-431.

[2] R. Dyer, H. Rajan, H. A. Nguyen, T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of Java language features", in Proc. of ICSE, 2013, pp. 779-790.

[3] S. Thummalapenta and T. Xie, "Mining exception handling rules as sequence association rules", in Proc. of ICSE, 2013, pp. 496-506.

[4] T. Xie and S. Thummalapenta, "Making exceptions on exception handling", in Proc. of WEH, 2013, pp. 1-3.

[5] J. Bloch, "Effective Java", 2nd Edition (The Java Series), Addison-Wesley, 2008.

[6] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes", in Proc. of OOPSLA, 2004, pp. 419-431, 2004.

[7] A. Mockus and D. M. Weiss, "Predicting risk of software changes", Bell Labs Technical Journal, 5(2), 2000, pp. 169-180.

[8] B. Cabral and P. Marques, "Exception handling: a field study in Java and .NET", in Proc. of ECOOP, 2007, pp. 151-175.

[2]https://asaduzzamanparvez.wordpress.com/researchall/