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 of applications, allow them to continue the normal flow of of executions in the event of exceptions or report such events to developers. Although techniques, features and bad coding practices of exception handling have been discussed both in developer communities and in the literature, there is a marked lack of empirical evidence of how developers use exception handling in practice. In this paper we use 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 relationship to the experience of developers. Our result gives some interesting insights. For example, we found that bad exception handling coding practices are common in source code projects and regardless of the experience all developers use exception handling coding bad practices.

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 executions. To enable programmers to deal with such exceptional situations, modern programming languages have built-in support for exception handling. For example, Java programming language uses several language constructs (such as try, catch, finally and throw) to support exception handling. Code that might throw exceptions need to be enclosed by a **try** statement that can catch those exceptions. The **try** statement needs to be supported by **catch** and **finally** statements that can contain instructions to specify the actions need to be taken when an 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 exceptions, there is a marked lack of empirical evidence of how developers use exception handling in practice. In this paper we utilize 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 II describes the data set used in our study. Section III describes briefly explain exception handling technique in Java. Section IV presents our research questions including results of our empirical study. Section IV summarizes the related work and Section V concludes the paper.

2. DATA SET

The data set used in this study is the 2015 GitHub data set from Boa. These include all Java projects on the GitHub with at least one or more Git repositories. The GitHub data set represents 274k projects with 22 million revisions contributed 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 the 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 error (see Fig. 1). Checked exceptions are those exceptions that a well written Java application should handle and recover from. Code that anticipates checked exception must follow the catch or specify requirements in Java. The requirements are as follows. The code should enclosed by a **try** statement and must followed by exception handlers. If an exception occurs in the **try** block, the exception will be

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

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

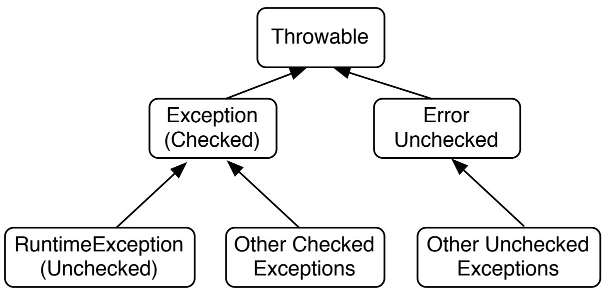

Figure 1: Exception Hierarchy in Java

handled by 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 *throws* clause in its declaration. The second kind of exception is the error. The causes of these exceptions are external to applications and that applications cannot recover form these exceptions. These are identified by **Error** and its subclasses. The last kind of exceptions 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 requirement. They are indicated by **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 incorrect use of exception handling we also investigate their relationships to the experience of developers, using new features, patterns in exception chaining and also in creating 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 to recover from exceptions.
- **Catching unchecked exception (CUE)**: Unchecked exceptions results of programming errors that can be fixed by checking proper conditions. Generally unchecked exceptions should not caught although there are exceptions to this idea.
- **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 exceptions. If they forget to wrap the original exception object within the new exception object they are throwing the exception by loosing the original source of the problem.

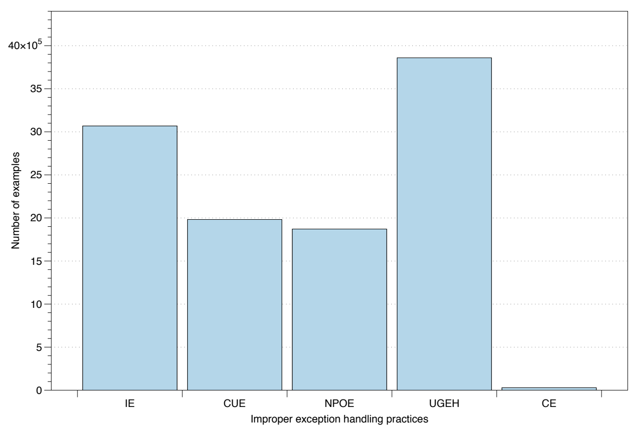

Figure 2: The frequency of different improper or bad exception handling coding practices

- **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.

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 GitHub data set. The figure shows that the data set contains significant number of all five bad coding practices. We find that using generic exception handler is the most frequent one. This is an indication that developers are very reluctant in using exception handling. Then comes the ignoring exceptions. Many developers are not aware of recovering from exceptions and thus leave the catch and finally blocks empty. This could cause difficulties maintaining applications as well as leave potential bugs in the code. We found a very small number of code examples where developers catch the error, indicates that most developers are aware of this bad practice.

4.2 How developers use exception chaining

In many applications lower level methods require to propagate information regarding exceptions to the higher level. This enables applications to notify end users about exceptions and users can take appropriate actions at the abstract level. Exception chaining is the mechanism that allows applications to propagate exceptions up the call stack and at the same time preserving important error information. To use exception chaining application uses *throw* statement that throws an exception in response to another exception. We are interested to investigate the patterns of throwing exceptions. The data can help other developers to learn various ways of constructing the **throw** statements. Table 1

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 ; }

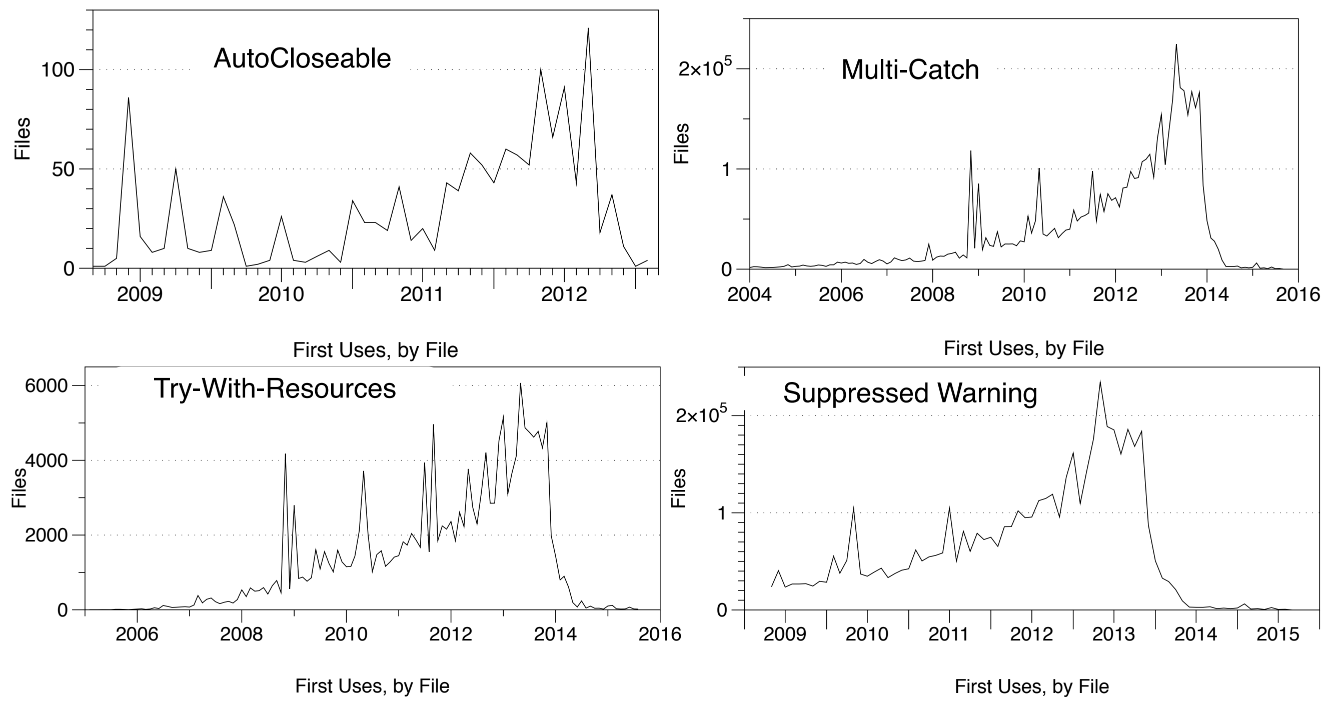

Figure 3: The number of files use new features for the first time over a period of time

shows different expression types developers used to throw exception in response to another exception. We see that the largest number of **throw** statements in exception chaining uses new expression type where they throw a new exception object that incorporates additional error message and may also contain reference to the lower level exceptions. Conditional expressions can be used to throw different expression objects with different error message information depending on some conditions. The assignment and null expressions are very infrequent. Expressions of variable access category become the second and the method call become the third most popular expression type used in exception chaining.

4.3 How 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 fulfil catch or specify requirement it may be the case that developers 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 how developers create their own exceptions. The data can indicate us which options developers prefer to use in practice. Interestingly when we investigate this in 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 **Runtime**, **Throwable** or **Error**, but those cases are very few in numbers.

4.4 When do developers use new exception handling features?

Java SE 7 comes with a number of changes in exception handling mechanism. These include catching multiple exceptions by using a single catch statement (also knows as multi-catch), try-with-resources declaration that frees developers to explicitly close resources, AutoCloseable interface

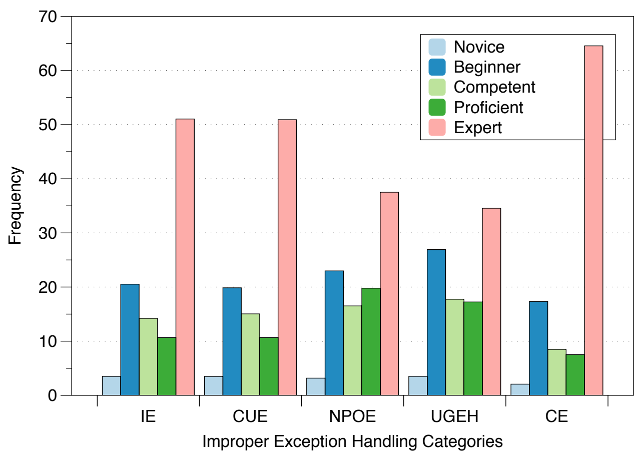

Figure 4: The frequency of different improper exception handling coding practices for various developer experience category

(classes that want to take advantage of try-with-resources need to implement the close method of the AutoCloseable interface) and suppressed warning. We are interested to find whether developers use these new features in their code. Fig 3 shows how many files incorporate a new features in each month. From the figure we can see that not only developers use the new features, but also developers start using these features long before their release (these features are officially released in 2011, July).

4.5 Do developers' experience affect exception handling coding practice

We are interested in finding how developers' experience affect bad exception handling coding practices. To calculate experience of developers, we use the total number of revisions committed to the source code by a developer up to a particular point of time. Here, in our analysis, we consider september 2015 as the particular point of time. We calculate the experience using the weighted mean of the number of commited revisions of all contributing developers as of Mockus and Weiss[7]. The weight is the proportion of the commit of a particular project and experience is her 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 5. Expert. Then, we investigate the improper coding practices among the above five experience categories. Result from our study (see Fig. 4) shows that expert developers are not concerned using the exception handling mechanism and they do not try to avoid the improper coding practices. We find out that, in most of the improper exception handling practices, they are the highest in using those patterns. This is an indication that they do not try to avoid those coding practices which could be problematic. Developers from other experience categories also use improper coding practices.

5. RELATED WORK

A number of studies have peen performed on exception handling. For example, Weimer and Necula [6] performed data flow analysis to find exception handling mistakes in resource management and characterize them. Cabral and Marques [8] analyzed 32 Java and .NET applications and found that exception handling are not used as an error handling tool. When exceptions occur applications take different recovery actions. Thummalapenta and Tie [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 how new exception handling features are introduced in the code. The most relevant study to ours is the work of Dyer et al. [2] that used Boa infrastructure to analyze the use of Java language features over time. While their work is to determine how new features are adopted once released in Java, we focus our attention to how developers use exception handling in Java.

6. CONCLUSION

In this paper we investigate exception handling in Java using the ultra large scale data set (274k Java projects) provided by Boa infrastructure. We use Boa language 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. Through our investigation on how developers use exception chaining and define their own exceptions, we find that when developers use them they typically follow the official Java language guidelines. We also investigate new exception handling features of Java. Our study also reveals that improper exception handling coding practices are not affected by the experience of developers and developers frequently use new features ahead of time. The programs and additional study results can be found online[2].

7. 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, T. Xie, “Mining exception handling rules as sequence association rules”, in Proc. of ICSE, 2013, pp. 496-506.

[4] T. Xie, 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. 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] Bruno Cabral, Paulo Marques, “Exception handling: a field study in Java and .NET”, in Proc. ECOOP, 2007, pp. 151-175.

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