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

Examining Programmer Practices for Locally Handling Exceptions

Mary Beth Kery, Claire Le Goues, Brad A. Myers
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
{mkery, clegoues, bam}@cs.cmu.edu

ABSTRACT

Many have argued that the current `try/catch` mechanism for handling exceptions in Java is flawed. A major complaint is that programmers often write minimal and low quality handlers. We used the Boa tool to examine a large number of Java projects on GitHub to provide empirical evidence about how programmers currently deal with exceptions. We found that programmers handle exceptions locally in `catch` blocks much of the time, rather than propagating by throwing an Exception. Programmers make heavy use of actions like Log, Print, Return, or Throw in `catch` blocks, and also frequently copy code between handlers. We found bad practices like empty catch blocks or catching `Exception` are indeed widespread. We discuss evidence that programmers may misjudge risk when catching `Exception`, and face a tension between handlers that directly address local program statement failure and handlers that consider the program-wide implications of an exception. Some of these issues might be addressed by future tools which autocomplete more complete handlers.

CCS Concepts

• **Repository Mining**

Keywords

Java Exceptions, Boa, GitHub, Error Handlers.

1. INTRODUCTION

When routine failures occur in a software system, exception-handling code is critical for the software to recover to a safe state or terminate safely. With Java checked exceptions, the compiler forces programmers to write code to deal with exceptions that can arise locally. In Java, the programmer can declare that an exception passes through this method by declaring it in the method header, forcing a higher-level program component in the call chain to handle it, or the programmer can handle the exception locally in `try/catch` block. The `try` block is used to surround at least one statement that may throw an exception. One or more `catch` blocks can follow a `try` block, each of which designates how the program will handle specific exceptions.

Programmers cannot ignore exception handling, yet past studies have argued that programmers do not write good-quality exception handlers [1][4]. A partial explanation is that there exists little support for programmers in this task. Several past studies [5][6] have introduced tools to help programmers understand exception propagation flow – that is, when the programmer wants to throw an exception or understand where a checked exception is coming from. These involve complex concerns, such as proper deallocation of resources on all possible error paths, or security concerns of implementation information that might be revealed on all possible error paths. Our data shows that programmers are actually struggling to use exceptions as the Java language intended on the most basic issues, even before addressing higher-level concerns.

In our current study we examined Java code at a large scale, over 11 million `try/catch` blocks from GitHub. We help illustrate that propagation is only one piece of the challenge programmers face. The majority of the time programmers do not themselves throw exceptions up the call chain, but locally handle exceptions raised by the methods they call. Simple bad practices are *extremely* prevalent. We found 12% of catch blocks were completely empty. Meanwhile, a full quarter of all exceptions caught are simply `Exception`. Note that although there are specific program situations where each of these bad practices can be reasonable, it is unlikely these explain the high prevalence we observed.

In this project, we focus on what `try/catch` blocks contain, and the decisions programmers appear to make when locally handling exceptions. We categorize the contents of exception handlers on GitHub. We find that the majority of instances are only a few lines of code, and much of the content can be described by simple actions like Throw, Log, Return, or print. We then discuss implications for aiding programmers with error handling through tool support.

2. RELATED WORK

Cabral et al. performed a closely related analysis to ours on 16 Java and 16 .Net programs, which were large production systems [4]. They manually examined exception handlers, so the descriptions in their work are more complete than ours, since we cannot determine the purpose of all code statements automatically. They categorized `catch` blocks into actions like “Log” or “Rollback”. Our work differs in that we examine a far larger set of Java projects and error handlers and focus on the attributes of a more general population of programmers. Code on GitHub ranges from large open-source projects down to student homework assignments. Our aim is to inform tool support that supports programmers in general with their exception handling needs.

MSR'16, May 14-15, 2016, Austin, TX, USA
 ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2903497

3. METHODOLOGY

3.1 Research Data

We analyzed nearly 8,000,000 Java repositories on GitHub provided by the Boa tool [1]. This contains 11,624,617 complete `try` exception-handling blocks. `Try/catch` blocks may be nested with `try/catch` blocks inside them, so here we count a "complete" exception handler as the outermost `try`. As a `try` can be followed by multiple `catch` blocks, the dataset contains more `catch` blocks (12,254,679) with an average of 1.05 `catch` blocks per `try`. While we focused our analyses on handlers with `catch` blocks, 14.2% of these `try` blocks have no `catch`, and re-throw any exceptions from the method declaration directly.

We did not include `try/catch` blocks found in test classes, which make up another 4 million complete handlers on GitHub. Program testing is a significant use for `try/catch` that is not strictly related to error handling. A test class runs a Java program on a variety of input values, and records which ones fail. Thus, in a test, catching *any* exception is acceptable, as is doing nothing but alerting the programmer of the failed test. These are excluded from the following analyses.

Java source code in the Boa tool is provided preprocessed into Abstract Syntax Tree (AST) form. Thus, `try/catch` blocks were easy to identify by their AST node label. Two drawbacks to the Boa format is that we lost any information about `finally` blocks (which were not encoded in Boa), and also any comments programmers left in the handlers, such as "//TODO" notes.

3.2 Exception Handler Categorization

AST nodes in Boa are labeled by their kind, such as METHOD, VARIABLE, or STATEMENT. For each repository and for each Java file, we recorded a description of every instance of a `try` or `catch` node. For each `catch` node we describe its child nodes (every program statement within that `catch` block) by their AST labels. Statements within `catch` are important because these are actions the programmer is taking to handle the exception. For certain statements within a catch block we record more detail. For THROW we record the type of the Exception thrown; for a RETURN we record the return argument. For EXPRESSION, we record the full AST information because EXPRESSION is used for method calls, including the method name and arguments. From method names, we were able to use simple text matching to refine our categorization of EXPRESSION into recognizable error-handling related methods such as a stack trace print or logging (a method whose name is "log" or "logger"). A limitation of simple text matching is that for most of the method calls we could not categorize their purpose. Future work may involve using more sophisticated methods such as from natural language processing.

In the AST, a `try` contains whatever code could go wrong, which is arbitrary for this analysis, so we record only nested `try/catch` or `throw` statements within a `try` block. Our final description for a complete exception handler was a sequence of labels in a single line, e.g., `"TRY, CATCH: SQLException: 1 line, EXPRESSION: "error()", END_CATCH, END_TRY`

4. RESULTS

4.1 Catch Block Size and Content

We used our representation of `try/catch` from Boa to describe the size (number of statements) and content of each `catch` block. While we observed a maximum size catch block of 888 statements, the vast majority of `catch` blocks contain only a few statements, as shown in Figure 1 on a log scale.

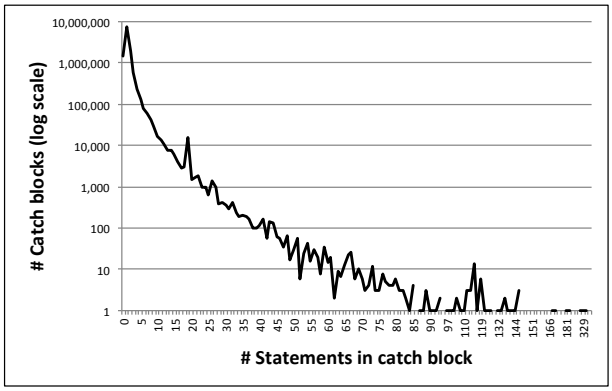

Figure 1: Frequency of `catch` blocks by number of statements in the `catch` block

There is an increase in the graph of `catch` blocks from 0 to 1 statements that is much shallower than we would hope to see. 12.4% (1,515,523) of the catch blocks have size 0 so they are literally empty and do nothing. These ignore and "swallow" exceptions (hide from the whole program that an exception occurred). There are situations where an empty `catch` is truly legitimate to the program logic, but it is generally considered bad practice [8] so seeing so this degree of them is worrisome.

In Figure 2, we show the distribution of content in a `catch` block of each size. We display the kinds of statements that occur in at least 1% of error handlers. For legibility, some statements are shown in broader categories – for instance "Control Flow" includes `if` statements, `for/while` loops, and `switch` statements. Control Flow (green in Figure 2) substantially increases with `catch` size.

"Print" is the second broad category of interest, as it includes normal print statements and also `e.printStackTrace()`. A programmer can print an exception's stack trace to get a detailed printout of the exception's propagation, which is useful for debugging However, having *only* `e.printStackTrace()` is generally bad practice because this is swallowing the exception. In our dataset, a full 10% of catch blocks print the stack trace and do nothing else. A generous hypothesis is that these programmers are at an early implementation stage where a debug console print is enough. We note also that this pattern, with a //TODO comment, is the default auto-complete for popular Java IDEs like Eclipse[1]. This suggests another hypothesis that programmers are simply leaving the default result of autocomplete.

Additionally, at least 10% of catch blocks only write to a log. We categorize methods as "log" if they include the word "log" and as "print" if they have the word "print," but these are not mutually exclusive, since some print methods can be specially configured to write to a log. We cannot easily automatically detect the configuration of a "print," so 10% is a conservative estimate that only counts "log". Logging takes no action to actually resolve a failure, but is acceptable in cases where the exception raised may have no real consequences for program state. Logging is an improvement over printing to the console because it records the exception in a permanent, reviewable form.

[1] https://eclipse.org/

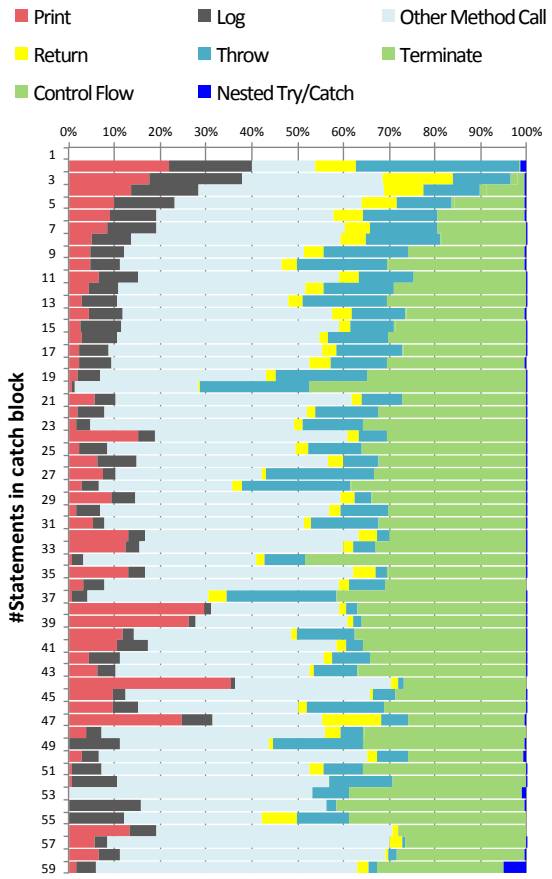

Figure 2: Kinds of statements in a catch block by the catch block's size.

Another 5% of catch blocks contain only `return`. We explored return statements, because exceptions were originally proposed as a language mechanism to avoid the ambiguity of error-code return values [7]. Exactly half of values returned from a catch block were `false` or `null`. These values often indicate a negative result in normal control flow. Like logging, this suggests programmers (whether correctly or not) deem a specific exception to have no real consequences on state, and present it instead as an un-exceptional failure. The remainder of returns could not be automatically categorized.

For `catch` blocks under 10 statements, which accounts for 11,981,327 or 98% of all `catch` blocks, the majority of actions are Throw, Return, Print, Log, and/or a method call.

4.2 Similarity Measures

Common actions like Throw, Return, Print, or Log are composable and simple. The error-handling quality of any of these is debatable based on a specific program's logic, but we suggest that these actions are evidence of exception handler *policies:* routine ways programmers understand to deal with exceptions. We further explored indicators of policy, or simply, repeated patterns in a program's `catch` blocks. How often do programmers not only use similar actions, but also repeat the *exact same code*?

We adapted the Levenshtein distance metric to estimate the similarity of two catch blocks. Levenshtein distance can be thought of as the minimum number of single-character edits (insertions, deletions, substitutions) required to change one string into the other. Given our string representation of a catch block, we treat each token label (e.g. CATCH) in the string as an individual 'character' for the purpose of distance measurement. Thus, the string "return null" can be edited to be "return foo" with a single replacement ("foo" for "null"). This is consistent with approaches taken in other studies of lexical source code similarity [10].

We calculated the average similarity between two `catch` blocks that are: in the same project, in the same file, and unrelated. The baseline average similarity of two `catch` blocks that are randomly sampled from different projects was 7%. The average similarity of `catch` blocks in the same project was 18%. However, the similarity of handlers in the same file is 65%, which is much higher than we would expect for generic code, inviting several hypotheses. A Java class generally contains related functionality, so intuitively it makes sense that similar exceptional situations will occur in a single class and be handled in similar ways. Thus using the same code may be reasonable much of the time. A negative view is that programmers may just be lazy, but we suggest programmers may be copying series of actions to try to reason about exception handling at a broader level than individual `catch` blocks. We suggest tool designers leverage these shared practices to offer programmers better suggestions or autocomplete for handler policies, e.g. always log when catching a certain kind of exception. Improved support may nudge programmers away from leaving empty catch blocks or ones that swallow exceptions.

4.3 Distribution of Exceptions Caught

Before handling exceptions, a programmer must decide which to catch. Given that programmers frequently take actions to dismiss exceptions, like only logging, returning, printing, or doing nothing, it is important that these decisions do not underestimate the risk that an exception may have on program state. We investigated another major bad practice: catching Java's top level `Exception` or `Throwable`. The danger in catching `Exception` is that while a programmer may be considering a simple local failure, the `catch` block will capture *all* checked exceptions that reach that program point. Catching `Throwable` causes the catch block to also handle runtime exceptions, including major system failures like `OutOfMemoryException`.

As shown in Figure 3, `Exception` and `Throwable` are both caught very often. `Exception` is a full 26% of all exceptions

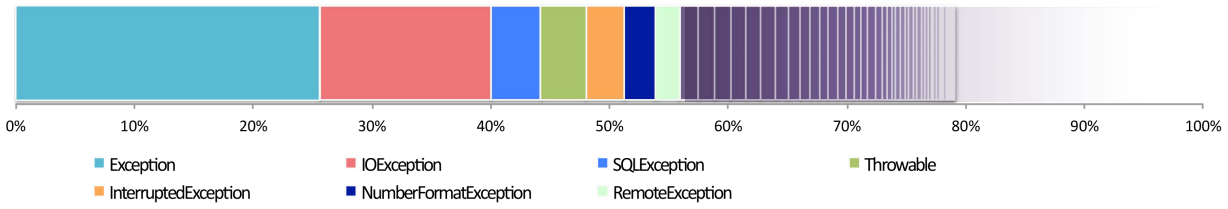

Figure 3: Exceptions caught by catch blocks on GitHub. Exceptions that occur more than 1% of the time are labeled. The rest, in purple, are thousands of exceptions that only rarely occur.

caught. To gain a sense of this usage, we manually examined a sample of 50 handlers that catch `Exception`. The most recurring program situation was contacting a web server or database and using `Exception` to cover any failures. Another pattern was catching `Exception` in the very last catch in a series of catch blocks, to cover any remaining unhandled errors. Many other examples used `Exception` to cover a specific failure, where a specific exception likely could have been used.

We propose several possible reasons for this catch-everything behavior. First, as shown in Figure 3, there are a few exceptions that are routine like `IOException`, which is typical for any I/O operations, or `InterruptedException`, which is typical for dealing with threads. Otherwise, given the diversity of thousands of highly specific and rarely occurring exceptions (purple in Figure 3), a programmer is faced with understanding the implications of exception classes they may rarely ever encounter. Some IDEs like Eclipse will auto-fill a `catch` with the correct exception classes thrown from a `try`. However `Exception` remains simple to remember and covers any failure.

A related explanation is that `Exception` is also a simple "umbrella" alternative for catching multiple exceptions. Our AST dataset was limited in that we cannot easily determine which exceptions are *possible* in a given `try` block. Thus, we explored this question in a limited way by looking at the occurrence of programmers catching multiple exceptions for a single try block. To catch multiple exceptions, the programmer has a choice: assign multiple `catch` blocks to the `try` to handle each separately, or handle multiple exceptions in the same catch by separating them with a '|'. The second option avoids redundant handler code, but is an addition since Java SE 7 in 2011 [11]. It is still rare, occurring in only 0.2% of catch blocks, so we hypothesize that some behavior catching `Exception` or duplicating catch blocks may be older, more familiar work-arounds for handling several exceptions in the same way. Most `try/catch` structures are one `try` per `catch`, but in 12% of `catch` blocks we observed more than one `catch` block in sequence, to handle exceptions separately.

We suggest tool support can help programmers choose what is appropriate to catch, and also appropriately group multiple exceptions. An issue arising with multiple exceptions is that they can be split into catch blocks differently by inheritance (e.g. all I/O child exceptions under `IOException`) *or* loosely split by the line in `try` that originally failed. Understanding failure by each line may often make sense, but is currently ambiguous for the programmer. Catching `RemoteException`, for example, has no obvious reference to which program statement caused it, without consulting documentation. Making this relationship between statements and their exceptions more visible with tools may help programmers avoid "umbrella" catches like `Exception`.

4.4 Local Throws

We have so far discussed the content of `catch` blocks, and the exceptions covered by them. A minority of `try` blocks result in an exception being propagated forward. In 14% of `try`, an exception is implicitly thrown by declaring it in the method signature only. In another 24% of `try`, an exception is thrown by method declarations mixed with explicit throws in `catch` blocks. Explicit throws are a surprising case of very positive programmer behavior. In 80% of cases, programmers re-cast exceptions before throwing, a practice recommended for better security to hide implementation details and maintain an appropriate abstraction [12]. Popular exceptions that programmers re-cast to are related to informing the caller of bad input: `IllegalArgumentException`, or `AssertionError`.

One caveat is that re-casting to `RuntimeException` is also common and is used by a fifth of re-casts. This abstracts away the real cause of an exception but gives callers no information about the failure. Catching `RuntimeException` can be something else entirely: a way to circumvent the Java checked exception system by misusing unchecked runtime exceptions [11].

5. CONCLUSION

We have demonstrated, through a large-scale analysis of `try/catch` blocks on GitHub, typical practices programmers use to handle exceptions. Future work will investigate leveraging the prevalence of common error handling actions for support tools to help suggest more positive handling policies to programmers.

6. ACKNOWLEGEMENTS

This research was funded in part by the NSF under grants CNS-1423054 and IIS-1314356 and the Air Force under Contract #FA8750-15-2-0075. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the US Government.

6. REFERENCES

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

[2] H B Shah, C Gorg, and M J Harrold. 2010. Understanding Exception Handling: Viewpoints of Novices and Experts. IEEE Transactions on Software Engineering 36, 2: 150–161.

[3] F Ebert and F Castor. 2013. A Study on Developers' Perceptions about Exception Handling Bugs. 2013 ICSM: 448–451.

[4] B Cabral and P Marques. 2007. Exception Handling: A Field Study in Java and .NET. ECOOP 4609, Chapter 8: 151–175.

[5] D Malayeri and J Aldrich. 2006. Practical Exception Specifications. Advanced Topics in Exception Handling Techniques 4119, Chapter 11: 200–220.

[6] H Shah, C Görg, and M J Harrold. 2008. Visualization of exception handling constructs to support program understanding. ACM, New York, New York, USA.

[7] J Goodenough. 1975. Exception handling: issues and a proposed notation. Communications of the ACM 18, 12: 683–696.

[8] IBM "Best Practice: Catching and re-throwing Java Exceptions" 01.ibm.com/support/docview.wss?uid=swg21386753

[9] B Goetz "Java theory and practice: The exceptions debate" www.ibm.com/developerworks/library/j-jtp05254/

[10] M Gabel and Z Su. 2010. A study of the uniqueness of source code. SIGSOFT FSE: 147–156.

[11] "Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking" docs.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html

[12] "Secure Coding Guidelines for Java SE" www.oracle.com/technetwork/java/seccodeguide-139067.html