CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Concordia University Research Repository

Studying and Assisting the Practice of Java and C# Exception Handling

Guilherme Bicalho de Pádua

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master of Applied Science (Software Engineering) at
Concordia University
Montréal, Québec, Canada

February 2018

© Guilherme Bicalho de Pádua, 2018

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Guilherme Bicalho de Pádua**

Entitled: **Studying and Assisting the Practice of Java and C# Exception Handling**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair

______________________________ Examiner
Dr. Nikolaos Tsantalis

______________________________ Examiner
Dr. Yann-Gaël Guéhéneuc

______________________________ Supervisor
Dr. Weiyi Shang

Approved by ______________________________
Dr Volker Haarslev, Graduate Program Director

TBD ______________________________
Dr Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Studying and Assisting the Practice of Java and C# Exception Handling

Guilherme Bicalho de Pádua

Modern programming languages, such as Java and C#, typically provide features that handle exceptions. These features separate error-handling code from regular source code and aim to assist in the practice of software comprehension and maintenance. Nevertheless, their misuse can still cause reliability degradation or even catastrophic software failures. Prior studies on exception handling aim to understand the practices of exception handling in its different components, such as the origin of the exceptions and the handling code of the exceptions. Previous research presented anti-patterns of exception handling; while little knowledge was shared about the prevalence of these anti-patterns. Furthermore, little is known about the relationship between exception handling practices and software quality. In this thesis, to complement prior research findings on exception handling, we, first, study the exception handling features by enriching the knowledge of handling code with a flow analysis of exceptions. Second, we investigate the prevalence of exception-handling anti-patterns. Finally, we investigate the relationship between software quality (measured by the chance of having post-release defects) and: (i) exception flow characteristics and (ii) 17 exception handling anti-patterns. Our case study is conducted with over 10K exception handling blocks, and over 77K related exception flows from 16 open-source Java and C# (.NET) libraries and applications. We collected a thorough list of exception flow characteristics and their anti-patterns using an automated exception flow analysis tool. On three Java and C# projects, we built statistical models of the chance of post-release defects using traditional software metrics and metrics that are associated with exception handling practice. We study whether exception flow characteristics and exception handling anti-patterns have a statistically significant relationship with post-release defects. Our case study results show that each try block has up to 12 possible potentially recoverable yet propagated exceptions. More importantly, 22% of the distinct possible exceptions can be traced back to multiple methods (average of 1.39 and max of 34). Moreover, we found that although exception handling anti-patterns widely exist in all of our subjects, only a few anti-patterns (e.g. *Unhandled Exceptions*, *Catch Generic*, *Unreachable Handler*, *Over-catch*, and *Destructive Wrapping*) can be commonly identified. Finally, we conclude that exception flow characteristics in Java projects have a significant relationship with post-release defects and there exist anti-patterns that can provide significant explanatory power to the chance of post-release defects. Our findings highlight the opportunities of leveraging automated

software analysis to assist in exception handling practices and signify the need for more further in-depth studies on exception handling practice. Development teams should consider allocating more resources to improving their exception handling practices and avoid the anti-patterns that are found to have a relationship with post-release defects.

Acknowledgments

In this short space of acknowledgments of people and the opportunities or support they provided me, I would like to express my gratitude and how much they represent to this work. It is a short space, nevertheless, nothing that this thesis represents, neither all the work that was put into it, could have been done without them.

First, I would like to thank Dr. Weiyi Shang, my supervisor. This work started because of his interesting research-oriented course. Since then, he has been always present by immediately answering all my questions and providing above and beyond support. Moreover, his feedback, guidance and brilliant ideas contributed enormously to the success of this work.

As this journey achieves its end, I'm grateful to have Dr. Nikolaos Tsantalis and Dr. Yann-Gaël Guéhéneuc as readers and evaluators. It is not only due to this thesis evaluation that Dr. Tsantalis, Dr. Guéhéneuc, and also other professors and students of the Software Engineering group that I'm thankful. Throughout the whole journey, I was always inspired by their work, as well as their ability to positively support all students in the group, towards improving our software engineering research. To that extent, I also thank the Department of Computer Science and Software Engineering.

I'm very thankful to my previous employer ERA Environmental, lead by Sarah Sajedi. ERA's support during the initial phases of my Master's was essential and I could not have come this far without it. I'm grateful for the years of work and experience we shared, and how they contributed to this work.

To not forget, I'm grateful to Canada. Through Canada's support via the Natural Sciences and Engineering Research Council of Canada (NSERC), I was awarded the Canada Graduate Scholarships-Master's Program (CGS-M). Such scholarship and recognition made me go above and beyond to achieve better research results. I'm also grateful for other Canadian researchers that through the Consortium for Software Engineering Research (CSER) provided valuable feedback to this work.

I'm thankful for the software engineering community. The kind volunteer work of peer reviewers improves our work by bringing the outsider view, reminding myself to put things in perspective and adjust our direction where needed.

Closer to my daily life outside of the scope of this work, I'm thankful for the SENSE lab mates and the DAS lab members. Together, we shared many moments in our research endeavours. I could not do it without the simple daily things we shared or the more intense experiences during conferences, seminars, courses and meetings.

Some close special people were always there for me during this whole process. My dear friends Stefan, Supreet, David and Kavijit were always open to listen and provide kind words of support. My parents, Selma and Oto, who always guided me and helped me stick to my own goals.

Finally, more than anyone else, I would like to thank my beloved fiancée Laura Weinkam. Laura's unconditional love, support, care and encouragement made me have the best of times during this academic experience. You were an inspiration to work hard, give my best and, more than anything, enjoy all parts of this learning experience. I dedicate this thesis to you.

Related Publications

The following publications are related to this thesis:

- *Chapter 3:* **Bicalho de Padua G**, Shang W. (2017). Revisiting Exception Handling Practices with Exception Flow Analysis. 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM) (11-20)
- *Chapter 4:* **Bicalho de Padua G**, Shang W. (2017). Studying the Prevalence of Exception Handling Anti-Patterns. 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC) (328-331). **Best ERA Paper Award**.
- *Chapter 5:* **Bicalho de Padua G**, Shang W. (2018). Studying the Relationship between Exception Handling Practices and Post-release Defects. 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR) (Submitted)

Contents

List of Figures xi

List of Tables xii

1 Introduction 1

1.1 Research Hypothesis . . . 2

1.2 Thesis overview . . . 2

1.2.1 Chapter 2: Background and Literature Review . . . 2

1.2.2 Chapter 3: Revisiting Exception Handling Practices with Exception Flow Analysis . . . 3

1.2.3 Chapter 4: Studying the Prevalence of Exception Handling Anti-Patterns . . . 4

1.2.4 Chapter 5: Studying the Relationship between Exception Handling Practices and Post-release Defects . . . 4

1.3 Thesis contributions . . . 5

1.4 Thesis organization . . . 5

2 Background and Literature Review 6

2.1 An illustrative example of exception handling practices . . . 6

2.1.1 Handling possible exceptions . . . 6

2.1.2 Raising and propagating exceptions . . . 7

2.1.3 Documenting exceptions . . . 8

2.2 Empirical studies on exception handling practices . . . 8

2.3 Anti-patterns of exception handling . . . 9

2.4 Improving exception handling practices . . . 11

2.5 Software quality and defect modeling . . . 11

3 Revisiting Exception Handling Practices with Exception Flow Analysis 13

3.1 Introduction . . . 14

3.2 Methodology . . . 15
3.2.1 Exception flow analysis . . . 15
3.2.2 Subject projects . . . 18
3.3 Quantity of Exceptions . . . 18
3.3.1 All and propagated possible exceptions . . . 18
3.3.2 Potentially recoverable yet propagated exceptions . . . 20
3.4 Diversity of Exceptions . . . 21
3.5 Sources of Exceptions . . . 23
3.5.1 Multiple sources of the same exception . . . 23
3.5.2 Sources of exception documentation . . . 24
3.6 Exception Handling Strategies and Actions . . . 24
3.6.1 Exception handling strategies . . . 25
3.6.2 Exception handling actions . . . 27
3.7 Threats to Validity . . . 30
3.7.1 External validity. . . . 30
3.7.2 Internal validity. . . . 30
3.7.3 Construct validity. . . . 31
3.8 Conclusion . . . 31

4 Studying the Prevalence of Exception Handling Anti-Patterns 33
4.1 Introduction . . . 33
4.2 Methodology . . . 34
4.2.1 Subject projects . . . 34
4.2.2 Detecting exception handling anti-patterns . . . 35
4.3 The prevalence of exception handling anti-patterns . . . 35
4.4 The amount of exception flows . . . 38
4.5 Discussion . . . 38
4.6 Threats to validity . . . 40
4.6.1 External validity . . . 40
4.6.2 Internal validity . . . 41
4.6.3 Construct validity . . . 41
4.7 Conclusion . . . 41

5 Studying the Relationship between Exception Handling Practices and Post-release Defects 42
5.1 Introduction . . . 43

5.2 Case Study Design . . . 44
5.2.1 Research questions . . . 44
5.2.2 Subject projects . . . 45
5.2.3 Metrics . . . 45
5.2.4 Model construction . . . 52
5.2.5 Model analysis . . . 54
5.2.6 Preliminary results . . . 55
5.3 Case Study Results and Discussion . . . 56
5.4 Threats to Validity . . . 60
5.4.1 External validity . . . 60
5.4.2 Internal validity . . . 61
5.4.3 Construct validity . . . 61
5.5 Conclusion . . . 62

6 Conclusions and Future Work **64**
6.1 Conclusion . . . 64
6.2 Future Work . . . 65

Bibliography **67**

List of Figures

1 An illustrative example. 7

2 Number of possible exceptions per try block for each project broken down by Propagated and Potentially Recoverable, Propagated, and Total. 20

3 Quantity of identified possible exceptions per source of information. 25

4 Quantity of possible exceptions per try block per project by handling strategy. . . . 26

5 Percentage of possible exceptions that are handled using each type of action. 28

6 Examples of differences between Java and C# and between applications and libraries. Differences between Java and C# are significant based on Wilcoxon Rank Sum test (p-value $<$0.05). Based on the same test, all differences between applications and libraries are not statistically significant. 40

7 An overview of our modeling approach: model construction and model analysis. . . . 53

List of Tables

1 List of the detected anti-patterns. 10
2 Our findings and implications on exception handling practices: quantity and diversity. 15
3 Our findings and implications on exception handling practices: sources and handling strategies and actions. 16
4 An overview of the selected subject projects. 19
5 Total amount of distinct exception types and the percentages of distinct exception types that appear in different quantity of try blocks. 22
6 Percentage of distinct exceptions that are traced back to one, two or over two methods. 23
7 List of the detected actions. 27
8 List of top 10 common exception types in the studied projects. 30
9 Overview of the selected subject projects. 35
10 Percentage of affected catch per project per anti-pattern. 37
11 Percentage of affected throws per project per anti-pattern. 37
12 Distribution of affected catch blocks according to flow anti-patterns and the quantity of affected flows. 39
13 An overview of the subject projects. 46
14 Exception handling flow characteristics metrics: part one of three. The symbol †indicates the rows where each metric represents multiple metrics. 48
15 Exception handling flow characteristics metrics: part two of three. The symbol †indicates the rows where each metric represents multiple metrics. 49
16 Exception handling flow characteristics metrics: part three of three. The symbol †indicates the rows where each metric represents multiple metrics. 50
17 Exception handling anti-patterns metrics. The symbol †indicates the rows where each metric represents multiple metrics. 51
18 A summary of the fitted models' construction and analysis. 56

19 Significant metrics in the final models with Wald χ^2 and effect values. Effect is measured by setting a metric to 110% of its mean value, while the other metrics are kept at their mean values. A positive impact (i.e., direction $\nearrow$) means that higher values of the metric, higher chance of bugs. 63

Chapter 1

Introduction

Modern programming languages, such as Java and C#, typically provide exception handling features, such as throw statements and try-catch-finally blocks. These features separate error-handling code from regular source code and are leveraged widely in practice to support software comprehension and maintenance [MSR85, CCHW09].

Having acknowledged the advantages of exception handling features, their suboptimal usage can still cause catastrophic software failures, such as application crashes [YLZ14, KZP+13], or reliability degradation, such as information leakage [Car96, ZC14]. A large portion of systems has suffered from system crashes that were due to exceptions [Cri82]. Additionally, the importance of exception handling source code has been illustrated in prior research and surveys [BGB14, ECS15].

Prior studies aim to understand the practices of exception handling in its different components: exception sources and handling code [SCKB16]. Findings from recent empirical studies have advocated the suboptimal use of exception handling features in open-source software [NHT16, KLM16, AARS16, BCR+15]. These prior research findings imply the lack of a thorough understanding of the practice of exception handling.

Prior research has reported a slew of anti-patterns on exception handling [YLZ14, CCHW09, McC06, BGB14]. These anti-patterns describe the problematic exception handling source code that may exist in the entire life cycle of exceptions, i.e., the propagation of the exception, the flow of the exception and the handling of the exception. Although these anti-patterns are discussed in prior research [SCKB16], the prevalence of these anti-patterns is not studied in-depth.

Moreover, the suboptimal practices and anti-patterns might not share a relationship with software quality, and, if that is the case, it may provide evidence to explain the findings from prior studies. However, little is known about the existence of such relationship.

In this thesis, to complement prior research findings on exception handling, we, first, study the

exception handling features by enriching the knowledge of handling code with a flow analysis of exceptions. Second, we investigate the prevalence of exception-handling anti-patterns. Finally, we investigate the relationship between software quality (measured by the chance of having post-release defects) and: (i) exception flow characteristics and (ii) 17 exception handling anti-patterns.

The rest of this chapter is organized as follows: Section 1.1 present our research hypothesis. Section 1.2 describes an overview of our case studies. Section 1.3 presents the main contributions of our case studies. Section 1.4 presents the organization of the rest of this thesis.

1.1 Research Hypothesis

Previous research findings and our industrial experience lead us to formulate the following research hypothesis:

> *The previously observed findings of exception handling characteristics and anti-patterns were scattered and diverse. Yet, it is not clear if the suboptimal practices are prevalent in practice and if they are related to software quality. We hypothesize that some suboptimal practices might be prevalent in practice and they might be related to software quality.*

The goal of this thesis is to empirically explore this hypothesis by revisiting the exception handling practices, studying the prevalence of suboptimal practices and modeling their relationship with software quality. In particular, we mine software repositories using automated tools we developed.

Moreover, we aim to specify and indicate which practices can and should be supported by automated tools that assist developers when handling exceptions. Our experience shows that developers have numerous sources of recommendations which can be overwhelming. In the case of that some suboptimal practices are related to software quality, software teams might be able to focus their efforts on specific suboptimal practices using automated analysis tools. Our findings are expected to pave a path for further research and knowledge transference to industrial settings.

1.2 Thesis overview

1.2.1 Chapter 2: Background and Literature Review

This chapter presents relevant research exception handling practices, exception handling anti-patterns, improving exception handling practices and the prior research on software defect modeling.

Overall, the previous research reviewed general aspects of exception handling, such as exception handling practices with and without flow analysis (i.e., handler actions); exception handling anti-patterns and smells are proposed; exception handling quantity, complexity and user studies on

developer difficulties when dealing with exception handling; documentation of exception handling; exception handling defects and impact; exception handling code changes and evolution. All of the studies provide empirical evidence that unveils the existence of suboptimal exception handling practices.

On one hand, undesired practices, especially defined anti-patterns, or rules are proposed as indicators of suboptimal exception handling practices [YLZ14, SCKB16, CCHW09, McC06, BGB14]. On the other hand, some researchers propose specific improvements for exception handling, such as tools to understand exception flow; exception handling design and mechanisms changes were proposed; automation of exception handling. However, the undesired practices and anti-patterns were not empirically studied.

As we discuss software quality, we also evaluate prior research on software quality measurement. We review previous research that used basic product and process metrics, as well as other aspects of software engineering (e.g. software logging or code review) and post-release defects. More importantly, we present the previous research on software patterns and anti-patterns and their relationship with post-release defects.

1.2.2 Chapter 3: Revisiting Exception Handling Practices with Exception Flow Analysis

In this chapter, we re-visit exception handling practices by conducting an in-depth study on 16 open-source Java and C# libraries and applications. To understand and analyze the state-of-the-practice of exception handling in these projects, we perform source code analysis to track the flow of exceptions from the source of exceptions, through method invocations, to the attempting blocks of exceptions (try block) and the exception handling block (catch block). With such flow analysis, we extract information about exception handling practice in over 10K exception handling blocks, and over 77K related exception flows from the studied subject systems.Our case study focuses on four aspects of the exception handling practices: 1) the quantity of exceptions, 2) the diversity of exceptions, 3) the sources of exceptions and 4) the exception handling strategies and actions.

Our results confirm the challenge of composing quality exception handling code. For example, we find a considerable amount of potentially recoverable yet propagated exceptions. However, more importantly, we highlight the opportunities of leveraging our automated source code analysis to complement the information that is valuable for developers when handling exceptions. More in-depth analyses are needed to ensure and improve the quality and usefulness of exception handling in practice.

1.2.3 Chapter 4: Studying the Prevalence of Exception Handling Anti-Patterns

In this chapter, we extend our analysis from the previous chapter and we investigate the prevalence of exception handling anti-patterns in 16 open-source Java and C# applications and libraries. We find that all of the studied subjects have exception handling anti-patterns detected in their source code. Whereas only five anti-patterns (*Unhandled Exceptions*, *Catch Generic*, *Unreachable Handler*, *Over-catch*, and *Destructive Wrapping*) are prevalently observed, i.e., in median detected in over 20% of the catch blocks or throws statements in the subject systems. We observe that these anti-patterns are often associated with multiple flows of exception, leading to bigger impact and more challenging resolution of such anti-patterns. By further investigation, we find that programming languages (e.g., Java or C#) may have a relationship to the existence of anti-patterns, while we do not observe such relationship with the type of projects (e.g., application or library).

Our results imply that, despite the prior research on exception handling, there is still lacking a deep understanding of the practice of exception handling. More in-depth analyses are needed to ensure the quality and usefulness of exception handling in practice.

1.2.4 Chapter 5: Studying the Relationship between Exception Handling Practices and Post-release Defects

In this chapter, based on the previous chapters findings of suboptimal exception handling practices (i.e., anti-patterns and flow characteristics), we perform an empirical study of the relationship between exception handling practices and post-release defects (as a proxy to software quality). In particular, our case study is conducted on two open-source Java projects (Hadoop and Hibernate) and one open-source C# project (Umbraco). Through the case study results, we would like to answer the following two research questions:

RQ1: Do exception flow characteristics contribute to better explaining the chance of post-release defects?

RQ2: Do exception handling anti-patterns contribute to better explaining the chance of post-release defects?

We find that, in some project (e.g., Umbraco), we do not observe any statistically significant relationship between exception flow characteristics and post-release defects. However, in the other two Java projects, the suboptimal practices of exception handling (e.g, the ambiguity of possible exceptions) indeed have a statistically significant relationship with post-release defects. In addition, although the majority of the anti-patterns do not have a statistically significant relationship with post-release defects, four anti-patterns are observed to be statistically significant. More importantly,

these anti-patterns may be prevalent ones and may provide large explanatory power to the chance of post-release defects in the studied projects.

Our case study results imply the importance of avoiding suboptimal exception handling practices. Furthermore, although not all anti-patterns are shown to be harmful, developers should at least consider avoiding the ones that are found to have a relationship with post-release defects in this study. Our findings can be used as a guideline for avoiding suboptimal exception handling practices.

1.3 Thesis contributions

Our thesis highlights the importance of avoiding suboptimal exception handling practices and advocates the need for techniques that can improve exception handling in software development practice.

In particular, the contributions of our thesis are:

1. We design automated tools that recovers exception flows from both Java and C#.
2. We present empirical evidence to illustrate the challenges and complexity of exception handling in open-source systems.
3. We present empirical evidence of the prevalence of exception handling anti-patterns.
4. Our exception flow and anti-patterns analysis, as an automated tool, can already provide valuable information to assist developers better understand and make exception handling decisions.
5. Our thesis is the first work that empirically studies the relationship between exception handling practice and the chance of post-release defects.
6. Our results provide guidelines to practitioners for improving their exception handling practices.

1.4 Thesis organization

The rest of this thesis is organized as follows: Chapter 2 presents the background and literature review of this thesis. Chapter 3 presents a study in which we revisit the exception handling practices with exception flow analysis. Chapter 4 reveals the prevalence of exception handling anti-patterns. Chapter 5 models post-release defects based on the exception handling practices. Finally, Chapter 6 summarizes our work and proposes future work.

Chapter 2

Background and Literature Review

In this chapter, we present the background and the prior research that is related to this thesis. In particular, we present an illustrative example and the prior research on exception handling practices, exception handling anti-patterns, improving exception handling practices and the prior research on software defect modeling.

2.1 An illustrative example of exception handling practices

In this section, we explain an illustrative example that handles, raises and propagates exceptions (see Figure 1). The example also illustrates the means of documenting exceptions.

2.1.1 Handling possible exceptions

In this example, a developer would like to implement a method named A. The method A requires to execute method B. The developer, by other means, has the knowledge that B can face two issues: 1) having an invalid path as input and 2) I/O faults. Therefore, instead of executing as expected, method B would possibly throw two types of exceptions: *InvalidPathException* and *IOException*, which correspond to the two issues, respectively. To deal with the two possible exceptions in method B, the developer needs to either handle the exception, i.e., determine the alternative actions when such exception happens, or propagate the exception such that a different method would manage the issue. In our example, the developer decides to handle *InvalidPathException* only and to propagate *IOException*.

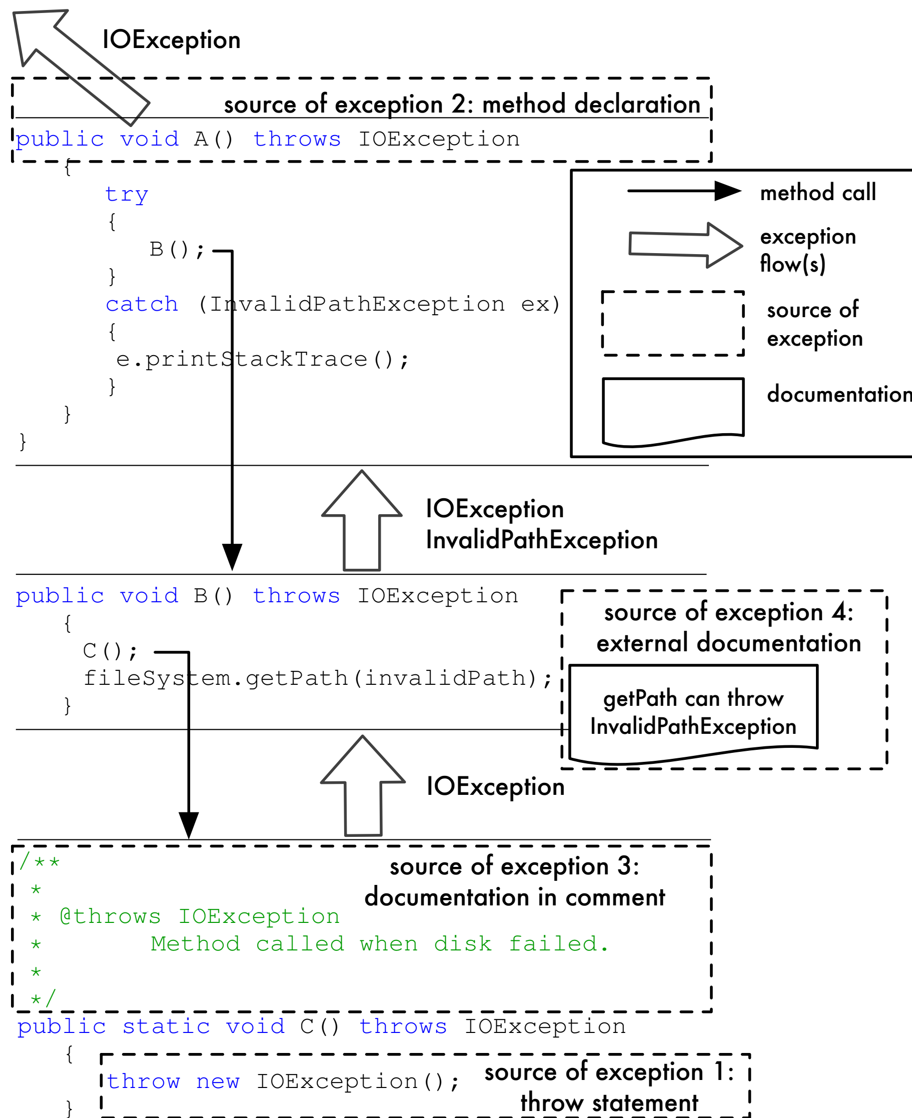


Figure 1: An illustrative example.

2.1.2 Raising and propagating exceptions

As mentioned, the developer knows that B can have two issues, corresponding to two possible exceptions. These two exceptions can be either newly raised or propagated from another method that B calls (e.g., method C and method *getPath*). Developers can use a throw statement to raise an exception. In our example, developers newly raise *IOException* with the throw statement in the method C. Moreover, if the issue happens, such exception will propagate to all the methods that call C.

2.1.3 Documenting exceptions

From our example, a developer could consult the source code of B and C to identify the rise of *IOException* in the throw statement. However, the source code is not always available for a method. For example, the method *getPath* called in method B is declared by an external API. Developers would need to consult the documentation (such as JavaDoc) of the method to discover the propagation of the possible *InvalidPathException*.

There could be cases when an exception is thrown but still not available in the documentation. Therefore, in some programming languages (like Java), a possible exception can be part of the method declaration. For example, method A, B, and C declares that a possible exception can propagate in the throws block of the method declaration. However, some exceptions can remain undeclared (like *InvalidPathException*).

2.2 Empirical studies on exception handling practices

Prior research studied exception handling based on source code and issue trackers. Cabral and Marques [CM07] studied exception handling practices from 32 projects in both Java and .Net without considering the flow of exceptions. Prior work by Jo *et al.* [JCYC04] focuses on uncaught exceptions of Java Checked exceptions. They proposed an inter-procedural analysis based on set-based framework without using declared exceptions.

Coelho *et al.* [CRG+08] assessed exception handling strategy with exception flows from Aspect-oriented systems and object-oriented systems. They evaluated the number of *uncaught* exceptions, exceptions caught by *subsumption*, and exceptions caught with *specialized* handlers.

Sena *et al.* [SCKB16] investigated sampled exception flows from 656 Java libraries for flow characteristics, handler actions, and handler strategies. We extend their work by looking into a higher number of flows per system (e.g. in Apache ANT we identified 930 catch blocks, compared to 2), by considering applications besides libraries and including C# .NET systems.

Some studies reveal that developers consider exception handling hard to learn and to use and tend to avoid it or misuse it [NHT16, KLM16, AARS16]. Bonifacio *et al.* [BCR+15] also surveyed C++ developers encountering revelations of educational issues. Asaduzzaman *et al.* [AARS16] found that regardless of their experience all developers exhibit improper exception handling coding practices. However, some improper exception handling categories, novice developers contribute the most.

It also has been noted that there is a lack of documentation of exceptions. Kechagia and Spinellis [KS14] found that 69% of the methods had undocumented exceptions and 19% of crashes could have been caused by insufficient documentation. Sena *et al.* [SCKB16]'s findings confirm that API

runtime exceptions are poorly documented. Cabral and Marques [CSM07] identify that infrastructure (20%) and libraries (15%) have better exception handling documentation when compared to applications (2%).

Significant research aimed to indicate exception handling problems and their impacts. Sinha *et al.* [SOH04] leveraged exception flow analyses to study the existence of 11 anti-patterns in four Java systems. Other research [CCHW09, ECS15, BGB14] classified exception-handling related bugs by mining software issue tracking. Thummalapenta and Xie [TX09] presented a rule-based approach and detected 160 defects, including 87 new defect not previously known, from 294 real exception-handling rules in five applications. Coelho *et al.* [CAG+17] mined Android stack traces and find a set of defect hazards related to exception handling anti-patterns, such as: cross-type wrappings, null pointer problems and undocumented runtime exceptions signalled by third-party code.

Cacho *et al.* [CCF+14, CBA+14] studied the evolution of the behavior of exception handling in Java and C# source code changes. Their results highlight the impact of the programming language design differences in the maintenance and robustness of exception handling mechanisms. Osman *et al.* [OCC+17] differentiates applications and libraries in terms of the usage of exception handling in an evolutionary study of Java systems. Oliveira *et al.* [OBS+18] studied Android software changes of regular code in comparison with changes in exception handling code. They found that the introduction of new Android-specific abstractions and invocations of methods of these abstractions are both very strongly correlated with an increase in the number of uncaught exception flows.

Our study revisits and combines different aspects of the studies mentioned above. Moreover, we present new findings that are not yet highlighted in prior research.

2.3 Anti-patterns of exception handling

There are different actions and their respective programming mechanisms involved in exception handling: 1) defining an exception using a type declaration, 2) raising an exception using a throw statement, 3) propagating an exception in a method by not handling it or using a throws statement and 4) handling an exception using a catch block. These mechanisms are illustrated also in an example on Section 2.1.

According to the implementation of the above actions, there can be different anti-patterns. In this thesis, we focus on the actions of propagation and handling of exceptions from the perspective of the explicit mechanisms (i.e. try-catch and throws). In particular, there exist three categories of related anti-patterns (see Table 1):

1. ***Flow*** anti-patterns are in the intersection of propagation (i.e. methods in the try block and its thrown exceptions) and handling actions (i.e. the catch block content) [YLZ14, MT97,

Table 1: List of the detected anti-patterns.

Group	Anti-pattern	Short Description
Flow	Over-catch	The handler catches multiple different lower-level exceptions [MT97, SCKB16].
	Over-catch and Abort	Besides over-catching, the handler aborts the system [YLZ14].
	Unhandled Exceptions	The handler does not catch all possible exceptions [SOH04].
	Unreachable Handler	The handler does not catch any possible exception [SOH04].
Handler	Catch and Do Nothing	The handler is empty [YLZ14, CCHW09, SOH04].
	Catch and Return Null	The handler contains return null [McC06, CCHW09].
	Catch Generic	The handler catches a generic exception type (e.g. Exception) [McC06, SOH04, MT97].
	Destructive Wrapping	The handler propagates the exception as a new exception [McC06]
	Dummy Handler	The handler only display or log some information [CCHW09].
	Ignoring InterruptedException	The handler catches InterruptedException and ignores it [McC06].
	Incomplete Implementation	The handler only contains TODO or FIXME comments [YLZ14].
	Log and Return Null	Besides being a dummy handler, the handler return null [McC06].
	Log and Throw	The handler logs some information and propagates the exception [McC06].
	Multi-Line Log	The handler divides log information into multiple log messages [McC06].
	Nested Try	The handler and its try block is enclosed in another try block [CCHW09].
	Relying on getCause()	The handler contains a call to getCause() [McC06].
	Throw within Finally	The handler is followed by a *finally* block that propagates exceptions [McC06].
Throws	Throws Generic	The throws propagates a generic exception type (e.g. Exception) [McC06].
	Throws Kitchen Sink	The throws propagates multiple exceptions [McC06].

SOH04, SCKB16].

2. ***Handler*** anti-patterns are only in the handling actions and are not related to the propagated exceptions [CCHW09, YLZ14, MT97, SOH04, McC06].

3. ***Throws*** anti-patterns are related to propagation issues, and they are specifically related to throws statement [SOH04, McC06].

This thesis is the first work to study the prevalence of exception handling anti-patterns extensively.

2.4 Improving exception handling practices

Robillard and Murphy [RM99] created a tool to analyze exception flows in Java programs, including a graphical user interface. Similarly, Garcia and Cacho [GC11] proposed a different approach for .NET related languages. Garcia and Cacho's tool supports visualization of metrics over the application history.

To support the software development lifecycle, Sinha *et al.* [SOH04] provided automated support for development, maintenance and testing requirements related to exception handling.

To improve how developers would deal with exception handling complexity, Kechagia *et al.* [KSS17] discuss and propose improvements in the design of exception handling mechanisms. Zhang and Krintz [ZK09] propose an as-if-serial exception handling mechanism for parallel programming. The programming languages also proposed new mechanisms to improve exception handling (e.g., try with resources [Rie11]) and previous research showed that they have been early-adopted, but the majority of the adoption was done in a later stage [AARS16].

The burden of writing exception handling code has been pointed out by Cabral and Marques [CM11]. They showed that a system with an automated set of recovery actions is capable of achieving better error resilience than a traditional system.

Barbosa *et al.* developed strategies with heuristics for recommending exception handling code as a semi-automated approach. Zhu *et al.* [ZHF+15] proposed an approach that suggests logging decisions for exception handling.

By conducting an in-depth study on 16 open-source projects, our findings illustrate the opportunities of leveraging various analysis to combine information from different sources to understand and assist in exception handling flows and practices. Our results are valuable to complement and assist in improving existing exception handling techniques.

However, it is still unclear if the observed and defined suboptimal exception handling practices are harmful, leading to bad software quality or whether the proposed analysis may improve the quality of software by improving exception handling. Therefore, in this chapter, we aim to study whether there exists a statistically significant relationship between the exception handling practices that are studied and defined in prior research, and the chance of having post-release defects, as one indicator of software quality.

2.5 Software quality and defect modeling

There exist a large body of research aiming to model software defects using product (e.g., the number of lines of code) and process metrics (e.g., the number of changes). Emam *et al.* [EEBGR01] revealed that size is a common confounding factor for the previously defined object-oriented metrics. In a

different work, D'Ambros *et al.* [DLR10] presented a benchmark for defect prediction comparison in terms of explanatory and predictive power of well-known defect prediction approaches (i.e., models with product and process metrics), together with novel approaches. Nevertheless, source code metrics are lightweight alternatives with overall good performance. In a comparison, Hassan [Has09] introduced change complexity metrics (e.g., number of prior faults) as indicators for future faults.

Besides basic product and process metrics, various research proposes metrics quantifying other aspects of software engineering in order to model software quality. For example, Shihab *et al.* [SBZ12] consider branching activities; Zhang *et al.* [ZKZH14] examine editing patterns, Shang *et al.* [SNH15] investigate logging characteristics and McIntosh *et al.* [MKAH16] study code reviews.

Moreover, researchers investigated the use of programming patterns and anti-patterns and their impact on software quality. Khomh *et al.* [KPGA12] and Taba *et al.* [TKZ+13] considered the use of anti-patterns because they are more actionable (e.g., developers can apply refactoring) than other metrics (e.g., churn). Their proposed anti-pattern based metrics provided additional explanatory power over the traditional metrics. Similar to this work, Khomh *et al.* [KPGA12] and Taba *et al.* [TKZ+13]: used logistic regression; tested which anti-patterns impact more and showed that size alone cannot explain defective classes. Moreover, Jaafar *et al.* [JGHK13] demonstrated that dependencies to classes with anti-patterns increase the chance of post-release defects.

To the best of our knowledge, this thesis is the first attempt to study the relationship between exception handling flow characteristics and their anti-patterns, and software quality. We base our study using the best traditional metrics from the afore-mentioned research that are shown to have a significant relationship with post-release defects.

Chapter 3

Revisiting Exception Handling Practices with Exception Flow Analysis

As presented in Chapter 2, Background and Literature Review, prior studies on exception handling aim to understand the practices of exception handling in its different components, such as the origin of the exceptions and the handling code of the exceptions. Yet, the observed findings were scattered and diverse. In this chapter, to complement prior research findings on exception handling, we study its features by enriching the knowledge of handling code with a flow analysis of exceptions. Our case study is conducted with over 10K exception handling blocks, and over 77K related exception flows from 16 open-source Java and C# (.NET) libraries and applications. Our case study results show that each try block has up to 12 possible potentially recoverable yet propagated exceptions. More importantly, 22% of the distinct possible exceptions can be traced back to multiple methods (average of 1.39 and max of 34). Such results highlight the additional challenge of composing quality exception handling code. To make it worse, we confirm that there is a lack of documentation of the possible exceptions and their sources. However, such critical information can be identified by exception flow analysis on well-documented API calls (e.g., JRE and .NET documentation). Finally, we observe different strategies in exception handling code between Java and C#. Our findings highlight the opportunities of leveraging automated software analysis to assist in exception handling practices and signify the need of more further in-depth studies on exception handling practice.

3.1 Introduction

Modern programming languages, such as Java and C#, typically provide exception handling features, such as throw statements and try-catch-finally blocks. These features separate error-handling code from regular source code and are leveraged widely in practice to support software comprehension and maintenance [MSR85, CCHW09].

Having acknowledged the advantages of exception handling features, their misuse can still cause catastrophic software failures, such as application crashes [YLZ14], or reliability degradation, such as information leakage [Car96, ZC14]. A large portion of systems has suffered from system crashes that were due to exceptions [Cri82]. Additionally, the importance of exception handling source code has been illustrated in prior research and surveys [BGB14, ECS15].

Prior studies on exception handling aim to understand the practices of exception handling in its different components: exception sources and handling code. Yet, the observed findings were scattered and diverse. Recent empirical studies on exception handling practices have advocated the suboptimal use of exception handling features in open source software [NHT16, KLM16, AARS16, BCR+15]. Moreover, in our previous research, we observe the prevalence of exception handling anti-patterns. These research findings imply the lack of a thorough understanding of the practice of exception handling.

Therefore, in this chapter, we re-visit exception handling practices by conducting an in-depth study on 16 open-source Java and C# libraries and applications. To understand and analyze the state-of-the-practice of exception handling in these projects, we perform source code analysis to track the flow of exceptions from the source of exceptions, through method invocations, to the attempting blocks of exceptions (try block) and the exception handling block (catch block). With such flow analysis, we extract information about exception handling practice in over 10K exception handling blocks, and over 77K related exception flows from the studied subject systems.

Our case study focuses on four aspects of the exception handling practices: 1) the quantity of exceptions, 2) the diversity of exceptions, 3) the sources of exceptions and 4) the exception handling strategies and actions. Tables 2 and 3 summarizes our findings and their corresponding implications. Such results confirm the challenge of composing quality exception handling code. For example, we find a considerable amount of potentially recoverable yet propagated exceptions. However, more importantly, we highlight the opportunities of leveraging our automated source code analysis to complement the information that is valuable for developers when handling exceptions. More in-depth analyses are needed to ensure and improve the quality and usefulness of exception handling in practice.

The rest of the chapter is organized as follows: Section 3.2 presents the methodology of the exception flow analysis through an illustrative example (i.e., Section 2.1) and our case study setup.

Table 2: Our findings and implications on exception handling practices: quantity and diversity.

Quantity of Exceptions (Section 3.3)	Implications
(1) There often exist multiple possible exceptions in each try block, and, out of those, many are propagated.	Current state-of-the-practice may not provide information to developers about all possible exceptions. Automated techniques may help developers be aware of all possible exceptions to make exception handling decisions.
(2) There exists a considerable amount of potentially recoverable exceptions that are propagated, even though they are recommended to be handled by Java and C#.	Exception flow analysis can provide automated tooling support to alert developers about not handling potentially recoverable exceptions.
Diversity of Exceptions (Section 3.4)	**Implications**
(3) With a significant amount of exceptions existing in each project, many possible exception types appear in only one try block.	Developers may not need to be aware of all exception types in a project by receiving automated suggestions of the exceptions that he/she needs to understand.

Section 3.3 to 3.6 presents the results of our case study. Section 3.7 discusses the threats to the validity of our findings. Finally, Section 3.8 concludes the chapter and discusses potential future research directions based on our research results.

3.2 Methodology

In this section, we present the methodology of our study. Aiding the explanation of our methodology, we first consider an illustrative example. Second, we introduce our exception flow analysis. Finally, we discuss the subject projects used.[1]

3.2.1 Exception flow analysis

In Section 2.1, we presented an example of an exception handling scenario with its related flows. In this section, we present our methodology that automatically extracts possible exceptions and their flows in Java and C# projects. We build an automated tool using Eclipse JDT Core and .NET Compiler Platform ("Roslyn") to parse the Java and C# source code, respectively. As an overview, our analysis consists of three main steps. First, we identify the exception handling blocks (catch blocks). Second, we recover the flow of exceptions by constructing the call graph that is relevant to

[1] Source code, binaries, statistical tests and Tableau visualizations with raw data are available online at https://guipadua.github.io/scam2017.

Table 3: Our findings and implications on exception handling practices: sources and handling strategies and actions.

Sources of Exceptions (Section 3.5)	Implications
(4) Over 22% of the exceptions are traced from different methods.	Automated tools are needed to help developers understand the source of the exception if it is traced back to different methods.
(5) The libraries used by the systems can provide documentation to most of the possible exceptions.	Developers should leverage automated analyses to understand possible exceptions.
Exception Handling Strategies and Actions (Section 3.6)	**Implications**
(6) Only a small portion of the exceptions are handled with the *Specific* strategy.	Developers should be guided to prioritize on handling exceptions with the *Specific* strategy, since developers cannot optimize the handling of the exception without knowing its exact type information.
(7) Java and C# have differences in leveraging various actions when handling exceptions.	More in-depth analysis and user studies are needed to further understand the rationale of differences of Java and C# exception handling practices.
(8) Actions that are taken when handling exceptions with specific or subsumption manners are not statistically significantly different.	Research and tooling support are needed to guide how to handle exceptions, especially with the specific strategy.
(9) With statistical significance, all top 10 Java and 2 out of top 10 C# exceptions have at least one action that is taken differently from the rest of the exceptions.	Developers may consider leveraging automated suggestions of exception handling actions.

the identified catch blocks. Finally, by traversing the flow of exceptions, we identify the sources of possible exceptions.

As a building block, we obtain the abstract syntax tree (AST) from the source code. In this step, we include not only the source code but also the binary files of dependencies from the Java Virtual Machine (JVM) for Java or the .NET Global Assembly Cache (GAC) for C#. The dependencies of third party libraries used by the projects are also included. These dependencies enrich the analysis by providing *binding* information, which draws connections between the different parts of a program (i.e. any method call and its origin, either if part of an internal declaration or external dependency). Also, we enrich the AST by parsing the documentation of the dependencies mentioned above as another source of information.

Identifying the handling of exceptions

We collect all the exception handling scenarios through all the catch blocks available in the AST. At the catch block, we use the AST elements to identify the methods that are executed to handle each exception as handling actions. We also obtain the related try blocks, which provide a list of called methods. These methods are necessary since they might raise or propagate the exceptions that the catch block potentially handles. In our example, we identify the catch block in method A. The method *printStackTrace* is the handling action of the exception *InvalidPathException*. From this catch block, we obtain the try block in which we find the call to method B. Method B can potentially propagate *InvalidPathException* and *IOException*.

Constructing call graph

Exceptions are propagated in method calls. Therefore, we leverage call graphs to recover the flow of exceptions. To handle polymorphism without risking over-estimation, we only consider the possible exceptions of the method that is declared in the parent class, since they are more generic and often called within the derived methods. Based on the previous step, for each identified method we traverse its call graph in a depth-first manner to find its possible exceptions. In our example, we traverse the call graph of method B and find two possible exceptions: *IOException* from method C and *InvalidPathException* from method *getPath*. Hence, based on the examples' catch block, we know that the *InvalidPathException* is handled in method A while *IOException* is propagated without handling.

Identifying sources of exceptions

During the call graph traverse and based on the AST, we identify four sources of exceptions. They are: 1) The newly raised exception by the throw statement, 2) the declared exception in the throws

of the method declaration (only for Java), 3) the documentation as comments in the source code (like JavaDoc comments), and 4) the external documentation. In our illustrative example, we can identify the newly raised *IOException* in a throw statement in method C (source 1), the declaration of the *IOException* in methods A, B, and C (source 2), and the JavaDoc documentation of method C for *IOException* (source 3). In addition, since we include the information from external libraries, our tool can also identify that the method *getPath* called in method B is a source of a possible *InvalidPathException* (source 4).

Some exceptions can be identified from multiple sources. For example, *IOException* is identified by three separate sources. We do not consider the multiple sources as different exceptions if the exceptions are associated with the same method call (e.g. method B). We label the separate sources of an exception as detailed information for each method call.

3.2.2 Subject projects

Table 4 depicts the studied subject projects. Our study considers Java and C# due to their popularity and prior research (see Section 2). Moreover, we include C# because of its different approach compared to Java exception handling. To facilitate replication of our work, we chose open-source projects that are available on GitHub.

We leverage GitHub filters on the number of contributors (i.e. projects with multiple contributors) and the number of stargazers (i.e. projects with more than ten stargazers), as they can achieve a good precision for selecting engineered software projects [MKCN17]. To narrow down the number of projects we also sorted the projects in descending order of the number of stargazers. Moreover, to potentially investigate the differences in exception handling practices and increase generalizability, we picked projects based on the filtering mentioned above. After reading the official description of the projects, we selected multiple applications and multiple libraries (i.e. project type), as well as multiple projects for different business domains (i.e. project purpose). From each project, we selected the most recent stable version of the source code at the moment of data collection for analysis.

3.3 Quantity of Exceptions

In this section, we study the quantity of all possible exceptions that are in each try block.

3.3.1 All and propagated possible exceptions

Ideally, developers should be aware of all possible exceptions to decide between handling or propagating them. To do that, developers need to navigate the call graph of a system that could extend

Table 4: An overview of the selected subject projects.

	Project	Release Version	Type	# Try	# Catch	# Method (K)	KLOC
C#	Glimpse	1.8.6	App.	56	57	1	31
	Google API	v1.15.0	Lib.	22	30	16	628
	OpenRA	release-20160508	App.	138	143	7	125
	ShareX	v11.1.0	App.	334	341	7	177
	SharpDevelop	5.0.0	App.	940	1,060	41	923
	SignalR	2.2.1	Lib.	94	105	2	38
	Umbraco-CMS	release-7.5.0	App.	595	615	15	362
Java	Apache ANT	rel/1.9.7	App.	934	1,139	11	158
	Eclipse JDT Core	I20160803-2000	Lib.	1,424	1,655	25	383
	Elasticsearch	v2.4.0	App.	385	408	12	108
	Guava	v19.0	Lib.	263	317	10	79
	Hadoop Common	rel/release-2.6.4	Lib.	975	1,144	14	147
	Hadoop HDFS	rel/release-2.6.4	App.	525	586	4	44
	Hadoop MapReduce	rel/release-2.6.4	App.	293	367	6	57
	Hadoop YARN	rel/release-2.6.4	Lib.	1,192	1,529	29	257
	Spring Framework	v4.3.2.RELEASE	Lib.	1,940	2,301	30	349
			Total	10,110	11,797	230	3,866

to multiple ramifications. Hence, the more exceptions there are, the more challenging (i.e. exponential growth) it is for developers to comprehend and decide about exception handling. Besides that, missing possible exceptions can be a reason for the lack of a handler that should exist, which is considered one of the top causes of exception handlings bugs [ECS15]. For those reasons, we study the quantity of total and propagated possible exceptions in each exception handling block.

As described in our methodology (see Section 3.2), we collect all the methods called in each try block. For those methods, we can recover the possible exceptions. Afterward, we can measure the quantity of possible exception by counting the unique types of exceptions in each try block.

We find that there typically exist multiple possible exceptions in each try block (see Figure 2). The median number of distinct possible exception per try block is four and two, for C# and Java respectively. More than 48% (C#) and 38% (Java) of try blocks can throw in between two and five exceptions. Moreover, more than 36% (C#) and 24% (Java) of the try blocks have six or more possible exceptions. For example, an important method named *processCompiledUnits(int,boolean)* in Eclipse JDT Core in the *Compiler* class has a try block with 33 distinct possible exceptions. Developers should properly handle exceptions in such an important method, to ensure reliability.

Among all possible exceptions, there often exist possible exceptions that are not handled by any

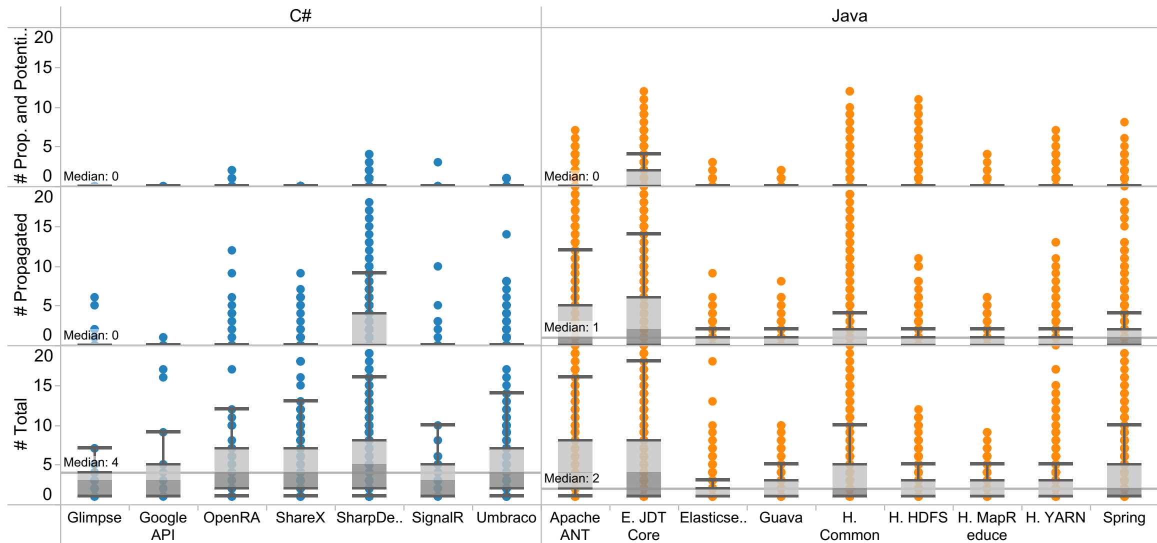


Figure 2: Number of possible exceptions per try block for each project broken down by Propagated and Potentially Recoverable, Propagated, and Total.

of the catch blocks that are associated with a try block [RM03]. These unhandled exceptions may increase the challenge of exception handling practice since they will be propagated and will need to be handled elsewhere. If the propagated exceptions remain uncaught across the whole system, there could be a risk of system failures [SOH04, BGB14, ECS15]. Therefore, we identify the possible exceptions that are not handled by the catch block.

Figure 2 presents the possible propagated exceptions for each try statement. We find that there may exist a large number (up to 34) of possible exceptions that are unhandled in each try block. For example, a method *execute(List,int)* from class *org.apache.tools.ant.taskdefs.optional.junit.JUnitTask* in Apache ANT has 25 possible exceptions while the corresponding catch block only handles *IOException*.

> **Finding 1:** There often exist multiple possible exceptions in each try block, and, out of those, many are propagated.
>
> **Implications:** Current state-of-the-practice may not provide information to developers about all possible exceptions. Automated techniques may help developers be aware of all possible exceptions to make exception handling decisions.

3.3.2 Potentially recoverable yet propagated exceptions

In the previous subsection, we find that a significant amount of the possible exceptions are propagated. However, not all exceptions are easy to recover or, more importantly, should even be recovered. For example, exceptions such as *ThreadDeath* in Java, and *OutOfMemoryException* in C#

cannot feasibly be recovered. In fact, both Java and C# define the recoverability level of exceptions in their documentation [GJS+15, .NE]. In particular, they suggest that developers should handle potentially recoverable exceptions while developers may not handle potentially unrecoverable ones. Hence, we first group all the propagated exceptions into either potentially recoverable or potentially unrecoverable, according to the specific guidance on Java or C# documentation. Then we count the number of propagated exceptions with potential recoverability.

We find that almost 8% (117) of C# and more than 19% (1,359) of Java try blocks have at least one potentially recoverable yet propagated exception. For example, a method named *rename* in Hadoop HDFS for file renaming features has a possible and potentially recoverable exception called *FileAlreadyExistsException*. This exception indicates the situation where a file is renamed to another existing file. However, this potentially recoverable exception is not handled by any catch block in that method.

Finding 2: There exists a considerable amount of potentially recoverable exceptions that are propagated, even though they are recommended to be handled by Java and C#.

Implications: Exception flow analysis can provide automated tooling support to alert developers about not handling potentially recoverable exceptions.

3.4 Diversity of Exceptions

There can be a diverse set of exceptions being used across try blocks. Prior research discusses that the use of a high number of distinct exception types might represent a greater concern with exception handling [BCR+15]. Therefore, in this section, we study the diversity of exceptions in our subject projects.

We count the total number of distinct exception types in each project, and the amount of try blocks in which each type of exception appears. Table 5 shows the percentage of the exception types of each project that appear in different quantities of try blocks. Despite the large number (up to 97 in C# and 249 in Java) of distinct exception types, there exist a considerable amount of exception types that only appear in few try blocks. In fact, over half of the exception types in C#, and almost 1/3 of the exception types in Java only appear in one try block. Such results imply that although the high number of distinct exception types may be a burden to developers, the burden may not be as high since a considerable amount of the exception types would only affect a small portion of the code.

Finding 3: With a large amount of exceptions exist in each project, many possible exception types appear in only one try block.

Implications: Developers may not need to be aware of all exception types in a project by receiving automated suggestions of the exceptions that he/she needs to understand.

Table 5: Total amount of distinct exception types and the percentages of distinct exception types that appear in different quantity of try blocks.

	Project	# Try blocks						Total
		1	2	3	4	5	>5	
C#	Glimpse	27.78%	38.89%	11.11%			22.22%	18
	Google API	28.00%	40.00%	12.00%		4.00%	16.00%	25
	OpenRA	35.71%	4.76%	7.14%	16.67%	2.38%	33.33%	42
	ShareX	13.04%	8.70%	6.52%	2.17%	6.52%	63.04%	46
	SharpDevelop	19.59%	8.25%	6.19%	4.12%	2.06%	59.79%	97
	SignalR	56.67%	16.67%	3.33%	6.67%		16.67%	30
	Umbraco	27.69%	9.23%	4.62%	1.54%		56.92%	65
	Total	55.88%	25.00%	13.97%	9.56%	5.15%	47.79%	214
Java	Apache ANT	15.73%	6.74%	6.74%	3.37%	3.37%	64.04%	89
	E. JDT Core	5.56%	2.78%	2.78%	4.17%	1.39%	83.33%	72
	Elasticsearch	27.78%	12.50%	16.67%	9.72%	4.17%	29.17%	72
	Guava	24.00%	12.00%	16.00%	2.00%	10.00%	36.00%	50
	H. Common	14.53%	15.12%	9.88%	11.63%	8.14%	40.70%	172
	H. HDFS	27.50%	13.75%	16.25%	7.50%	1.25%	33.75%	80
	H. MapReduce	21.74%	8.70%	4.35%	8.70%	2.17%	54.35%	46
	H. YARN	17.53%	4.12%	11.34%	6.19%	3.09%	57.73%	97
	Spring	22.09%	12.05%	7.63%	10.04%	4.82%	43.37%	249
	Total	32.93%	17.76%	15.97%	14.77%	8.38%	42.32%	662
Grand Total		37.83%	19.31%	15.54%	13.66%	7.69%	43.49%	876

3.5 Sources of Exceptions

The same exception may be traced back from different sources. In this section, we study the sources of exceptions per try block.

3.5.1 Multiple sources of the same exception

The multiple sources of exceptions may increase the complexity of exception handling. Consequently, a developer would need to comprehend and investigate more methods in the source code to effectively handle exceptions. For example, developers may encounter a *FileNotFoundException* due to missing an input file or configuration file. However, developers may need different actions to handle such an exception since missing an input file may be caused by users' mistake while missing a configuration file is a critical issue of the software. Multiple sources of the same exception may also impact testers since they would need to properly test the exception behavior as well as the multiple possible paths of control flow.

Table 6: Percentage of distinct exceptions that are traced back to one, two or over two methods.

	# Distinct methods				Total
	0	1	2	>2	
C#	1.05%	76.11%	14.05%	8.80%	7,638
Java	0.61%	77.18%	13.00%	9.20%	28,854

We group each possible exception by the distinct methods that act as a source of exceptions. We only consider distinct methods since the same method may not need different ways to handle the exception while the exception propagated from various methods may need to be handled differently. In Table 6, we present the percentage of possible exceptions that are traced back from zero, one, two and more than two distinct methods. The first group is from zero methods, which means that these possible exceptions were traced back to explicit throw invocations, not method invocations.

Although most of the possible exceptions (above 76%) are traced back to a single method, we observe that more than 22% of the exceptions are traced back to multiple invoked methods. The try blocks with the highest number of methods can have from two to 17 among C# projects; while, for Java, it is between five and 34. For example, the Umbraco C# class called *TypeFinder* performs lazy accesses to all assemblies inside a single try block and therefore *System.ArgumentNullException* can be traced back from 14 different invoked methods. We also noticed that exceptions that are super classes of other exceptions (e.g., *IOException*) have a higher than average chances of being from multiple sources.

Finding 4: Over 22% of the exceptions are traced from different methods.

Implications: Automated tools are needed to help developers understand the source of the exception if it is traced back to different methods.

3.5.2 Sources of exception documentation

Prior studies revealed that lacking immediate documentation is one of the challenges of exception handling [CM07, SCKB16, KS14]. Prior studies observed a small number of documented exceptions. As shown in our illustrative example (see Section 2.1), possible exceptions can be recovered from up to four different sources. They are: 1) the newly raised exception by the throw statement in the source code, 2) the declared exception in the throws of the method declaration (only for Java), 3) the documentation as comments in the source code (like JavaDoc comments) and 4) the external documentation. By recovering the sources of each possible exception using exception flow analysis, we may be able to provide the documentation of possible exceptions. For Java, we only recover documentation for unchecked exceptions since checked exceptions must be specified in method declarations.

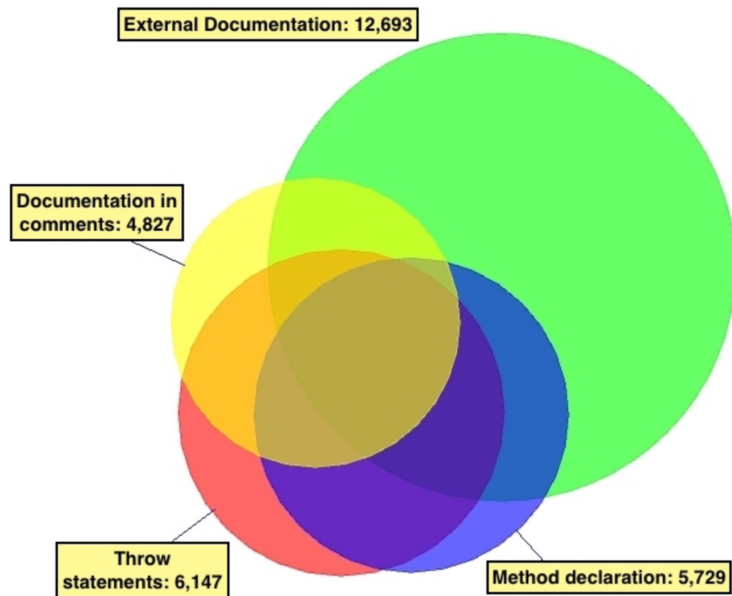
We find that, from all the possible exceptions that we identify, 93% for Java and 71% for C# can be retrieved from the external documentation of dependencies. Figure 3 depicts the sets of possible exceptions per try block that were retrieved by our exception flow analysis. Our findings show that the challenge of having a low amount of documented exceptions can be well addressed by applying exception flow analysis with the information from external documents of libraries. We find that such rich documents are typically from the exceptions that are provided by the system libraries. Therefore, the high availability of such documentation can be expected to assist developers not only for our subject systems but also for the majority of Java and C# projects.

Finding 5: The libraries used by the systems can provide documentation to most of the possible exceptions.

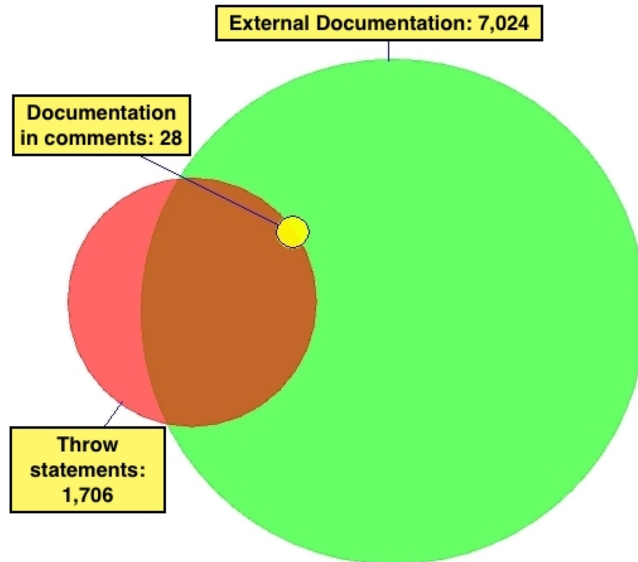
Implications: Developers should leverage automated flow analysis to understand possible exceptions.

3.6 Exception Handling Strategies and Actions

In this section, we study the strategy and actions in exception handling practices considering the exception flows of each handler.



(a) Java Unchecked Exceptions.



(b) C# Exceptions

Figure 3: Quantity of identified possible exceptions per source of information.

3.6.1 Exception handling strategies

Exception handling strategy describes the manner in which an exception is handled. In particular, the relationship between the possible exception in a try block and the handler exception in the corresponding catch blocks. There exist in total two handling strategies:

- *Specific*, is the strategy when the type of a possible exception is exactly the same as the handler

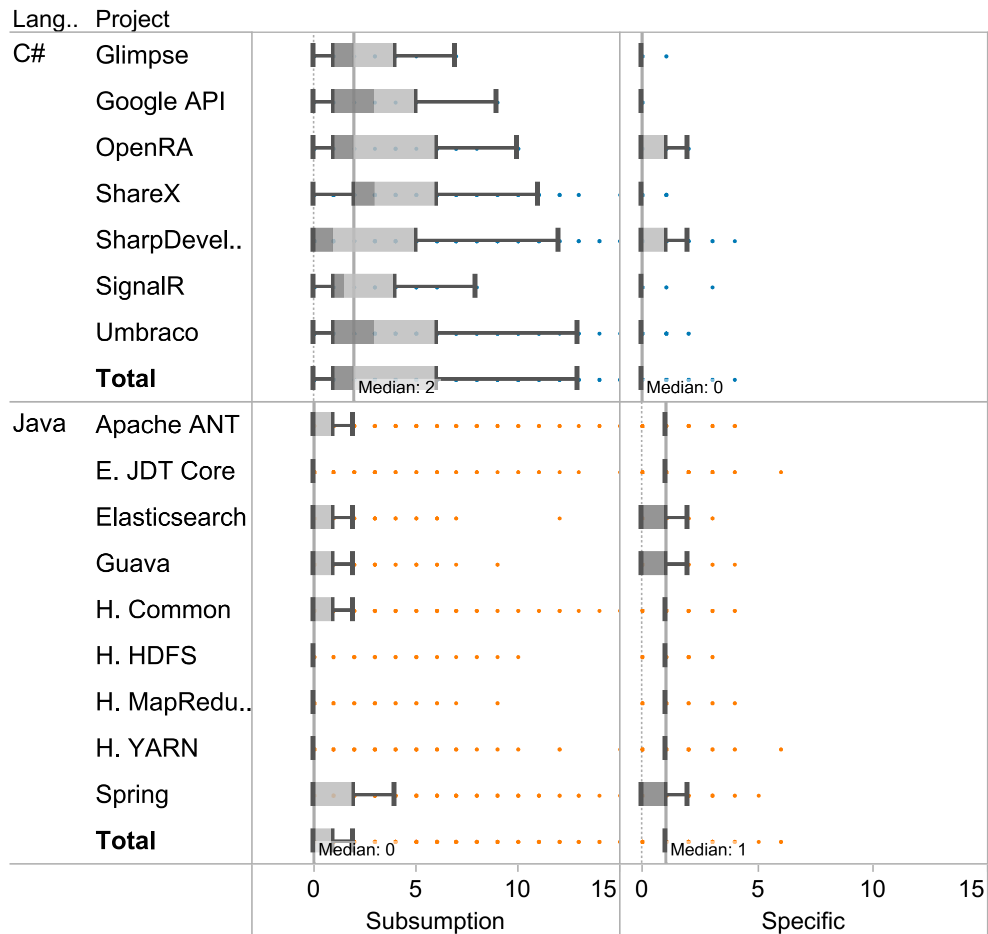


Figure 4: Quantity of possible exceptions per try block per project by handling strategy.

exception.

- *Subsumption*, is the strategy when the handler exception is a superclass of a possible exception.

Since there can be multiple possible exceptions, it can be overwhelming for a developer to handle each possible exception with a *Specific* strategy. On the other hand, the *Subsumption* strategy may introduce uncertainty to the caught exception. To study the handling strategies, we compare each possible exception with the handler exception in the corresponding catch block. Figure 4 depicts the quantity of distinct possible exception per try block that is handled according to each strategy.

The majority of the exceptions are handled with a subsumption strategy, while only a small portion of the exceptions are handled specifically. The results show that developers tend to over-catch exceptions. The extreme case of subsumption strategy is the "Catch Generic" exception handling anti-pattern [BdPS17b], where developers simply use an exception type which can catch any

exceptions in the software, e.g., *Exception* in Java. Such practice is heavily discussed in prior research [SOH04, SCKB16] and is considered to be harmful since developers cannot optimize the handling of the exception based on the exact type of the exception, but rather only know that there may exist some exceptions during run-time.

Finding 6: Only a small portion of the exceptions are handled with the *Specific* strategy.

Implications: Developers should be guided to prioritize on handling exception with the *Specific* strategy, since developers cannot optimize the handling of an exception without knowing its exact type information.

3.6.2 Exception handling actions

During the exception flow analysis (see Section 3.2), we collect a set of method calls in each catch block to know how each exception is handled. Prior studies [YLZ14, SCKB16, CM07, CRG+08, ZHF+15] propose a list of actions based on the combination of method calls in the catch block as Exception handling actions. Table 7 presents the list of actions that are defined in prior research and are used in this thesis. To further understand how exceptions are handled, we study the exception handling actions in our subject projects.

Table 7: List of the detected actions.

Action	Short Description
Abort	The handler contains an abort statement [YLZ14].
Continue	The handler contains a continue statement [CM07].
Default	The handler contains the IDE suggested method (Java only).
Empty	The handler is empty [YLZ14, SCKB16, CM07, CRG+08].
Log	The handler display or log some information [SCKB16, CM07, CRG+08].
Method	The handler contains a method invocation different than the other actions listed in this table. [CM07, SCKB16].
Nested Try	The handler contains a new try statement [ZHF+15].
Return	The handler contains a return statement [SCKB16, CM07].
Throw w/o New	The handler contains a throw statement without a new exception instantiation [SCKB16, CM07, CRG+08].
Throw New	The handler contains a throw statement with a new exception instantiation [SCKB16, CM07, CRG+08].
Throw Wrap	The handler contains a throw statement using the original exception or its associated information [CRG+08].
Todo	The handler contains TODO or FIXME comments [YLZ14].

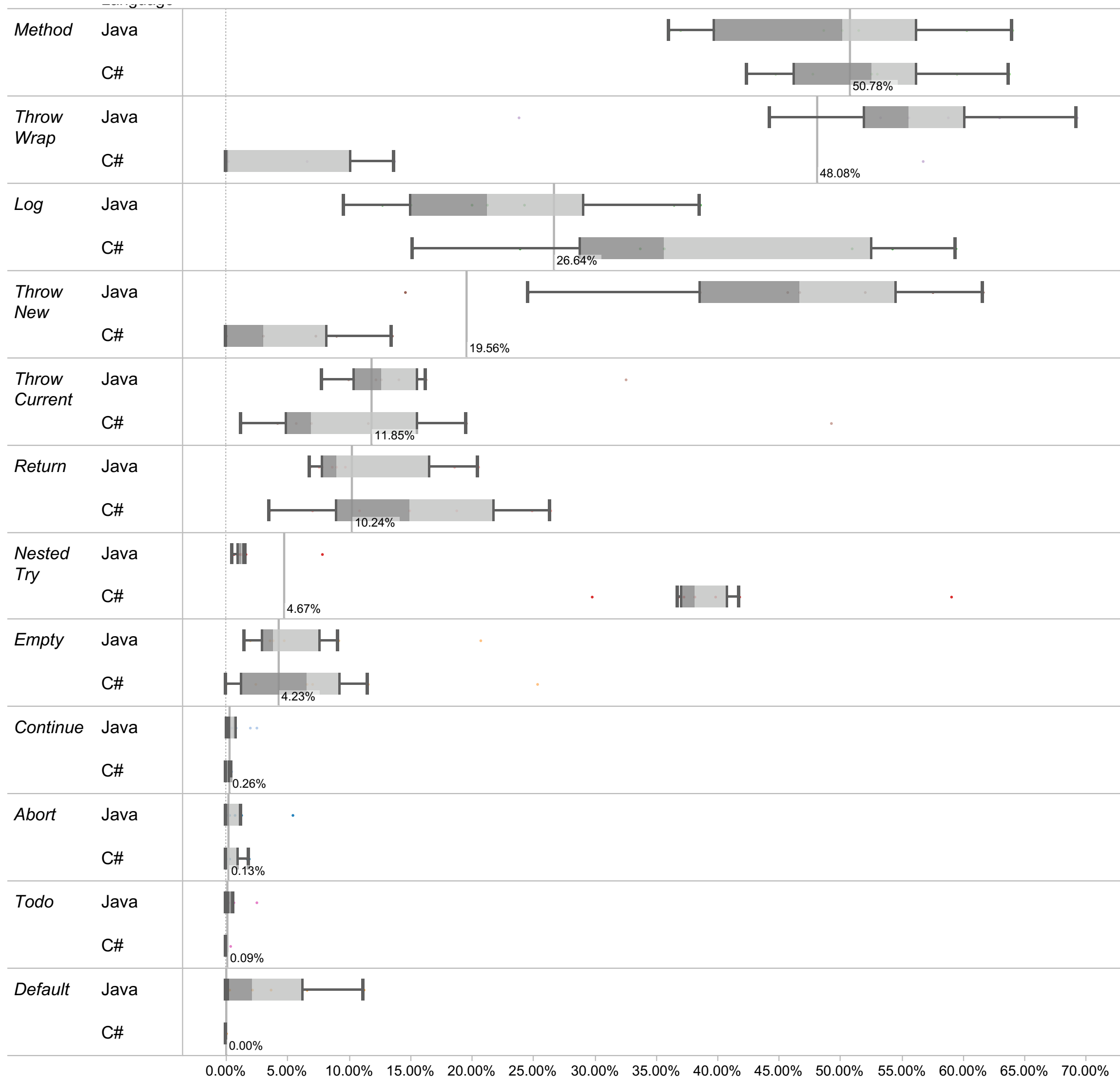


Figure 5: Percentage of possible exceptions that are handled using each type of action.

Figure 5 presents the percentages of possible exceptions of each project that are handled using a particular type of action. We observe that Java and C# have differences in executing various actions when handling exceptions. To determine the differences, we perform Wilcoxon Rank Sum test [WW64] to compare the percentage of possible exceptions that are handled using each type of action in each C# and Java project. Hence, we examined if there exists statistically significant difference (i.e. p-value < 0.05) between Java and C#. A p-value < 0.05 means that the difference is likely not by chance. We choose Wilcoxon Rank Sum test since it does not have an assumption on the distribution of the data.

We find statistically significant difference between Java and C# for exception handing with actions "Throw Wrap", "Throw New", "Nested Try", "Continue" and "Todo". Among these actions, "Throw Wrap", "Throw New"and "Todo" may indicate that the exceptions are not effectively handled but rather propagated or ignored. All these three actions show up more in Java than C#. We consider the reason may be that Java compiler forces developers to explicitly manage checked exceptions while developers may not have the knowledge of how to handle them properly. To simply make the program compile, developers potentially take these actions. Further studies should investigate why such actions are chosen more in Java than C#.

Finding 7: Java and C# have differences in leveraging various actions when handling exceptions.
Implications: More in-depth analysis and user studies are needed to further understand the rationale of differences of Java and C# exception handling practices.

We also compare the actions that are taken when the exceptions are handled with either specific or subsumption strategy. We perform Wilcoxon Rank Sum test similar as when comparing Java and C#. This time, for each particular programming language and type of action, the test compared the percentage of specific handling in each project with subsumption handling in each project. Nonetheless, we observe only one action, i.e., *Log*, that is handled differently (statistically significant) with specific or subsumption strategy.

Finding 8: Only one action, *Log* in Java, is taken differently when exceptions are handled with specific or subsumption strategy.
Implications: Research and tooling support are needed to guide how to handle exceptions, especially with the specific strategy.

We would like to know if any particular actions are taken when handling some special possible exception. With such knowledge, we may be able to suggest actions automatically to developers handling exceptions. We gather a list of the ten most handled types of possible exceptions in Java and C#, respectively (see Table 8). We also obtain the percentage of possible exceptions that are handled using each action for each exception type. Similarly, we use Wilcoxon Rank Sum test to compare. For each particular programming language, action, and exception type, the test compared the percentage of the given type in each project with the combined value of all other types of exceptions in each project. We find with statistical significance that, for Java, all top exceptions have at least one action that is taken differently from the rest of the exceptions, and, for C#, two top exceptions has such difference.

Table 8: List of top 10 common exception types in the studied projects.

C#	Java
System.ArgumentNullException	java.io.IOException†
System.ArgumentException	java.lang.IllegalArgumentException†
System.NotSupportedException†	java.lang.NullPointerException†
System.ArgumentOutOfRangeException	java.lang.IndexOutOfBoundsException†
System.InvalidOperationException	java.lang.SecurityException†
System.FormatException	java.lang.IllegalStateException†
System.IO.IOException	java.lang.ExceptionInInitializerError†
System.IO.PathTooLongException†	java.lang.ArrayStoreException†
System.Security.SecurityException	java.lang.IllegalAccessException†
System.ObjectDisposedException	java.lang.ClassNotFoundException†

†The exception has at least one actions that taken statistically significantly differently from the rest exceptions.

Finding 9: All top 10 Java and 2 out of top 10 C# exceptions have at least one action that is taken statistically significantly differently from the rest exceptions.

Implications: Developers may consider leveraging automated suggestions of exception handling actions.

3.7 Threats to Validity

In this section, we discuss the threats to validity of our findings.

3.7.1 External validity.

Our study is based on a set of open-source Java and C# projects from GitHub. Our findings may not generalize to other projects, languages or commercial systems. Replicating our study on other subjects may address this threat and further understand the state-of-the-practice of exception handling.

3.7.2 Internal validity.

We aim to include all possible sources of information in our automated exception flow analysis. However, our analysis may still miss possible exceptions, if there is a lack of documentation or the source code is not compilable. Also, the documentation of the exception may be incorrect or outdated. In our analysis, we trust the content of documentation. Therefore, we cannot claim that

our analysis fully recovers all possible exceptions nor that the recovered information is impeccable. Further studies may perform deeper analysis on the quality of exception handling documentation to address this threat.

Our study aims to consider the try-catch block since that is an actionable place for developers. For that reason, we study only exception flows that are under the scope of try-catch blocks. Future work might consider analyzing all other scopes (i.e., code outside try-catch blocks).

3.7.3 Construct validity.

Our study may not cover all possible handling actions. We selected actions based on the previous research in the subject [YLZ14, SCKB16, CM07, CRG+08, ZHF+15]. Some actions are not included in our study if they are either 1) require heuristic to detect or 2) are not well explained in details in related work. Moreover, we may not include differences due to newer features provided by the programming languages (e.g., try with resources [Rie11, AARS16]) might affect the results of our studies and such features might not be applicable in all programming languages.

Our possible exception identification approach is based on a call graph approximation from static code analysis. We may still miss possible exceptions due to under-estimation for polymorphism or unresolved method overload. Although such approximation may impact our findings, our choice of under-estimation would not significantly alter the existence of observed challenges of exception handling, i.e., the challenge may appear even worse without the under-estimation. Nevertheless, to complement our study, dynamic analysis on the exception flow may be carried out to understand the system exceptions during run-time.

3.8 Conclusion

Exception handling is an important feature in modern programming languages. However, prior studies unveil the suboptimal usage of exception handling features in practice. In this chapter, we revisit practice of exception handling in 16 open source software in Java and C#. Although we confirm that there exist suboptimal manners of exception handling, more importantly, we highlight the opportunities of performing source code analysis to recover exception flows to help practitioners tackle various complex issues of handling exceptions. In particular, the contributions of this chapter are:

1. We design an automated tool that recovers exception flows from both Java and C#.

2. We present empirical evidence to illustrate the challenges and complexity of exception handling in open source systems.

3. Our exception flow analysis, as an automated tool, can already provide valuable information to assist developers better understand and make exception handling decisions.

This chapter highlights the opportunities and urgency of providing automated tooling to help developers make exception handling decisions during the development of quality and reliable software systems.

Chapter 4

Studying the Prevalence of Exception Handling Anti-Patterns

After revisiting the exception handling practices with flow analysis and highlighting the suboptimal practices, in this chapter, we extend our exception flow analysis to investigate the prevalence of specific previously defined exception handling anti-patterns. Prior studies suggested anti-patterns of exception handling; while little knowledge was shared about the prevalence of these anti-patterns. In this chapter, we investigate the prevalence of exception-handling anti-patterns. We collected a thorough list of exception anti-patterns from 16 open-source Java and C# libraries and applications using an automated exception flow analysis tool. We found that although exception handling anti-patterns widely exist in all of our subjects, only a few anti-patterns (e.g. *Unhandled Exceptions*, *Catch Generic*, *Unreachable Handler*, *Over-catch*, and *Destructive Wrapping*) can be commonly identified. On the other hand, we find that the prevalence of anti-patterns illustrates differences between C# and Java. Our results call for further in-depth analyses on the exception handling practices across different languages.

4.1 Introduction

Exception handling features, such as throw statements and try-catch-finally blocks, are widely used in modern programming languages. These features separate error-handling code from regular code and are proven to enhance the practice of software reliability, comprehension, and maintenance [MSR85, CCHW09]. On the other hand, the misuse of exception handling features can cause catastrophic failures [YLZ14]. A prior study shows that two-thirds of the studied system crashes were due to exceptions [Cri82]. Barbosa *et al.* [BGB14] illustrate the importance of the quality of exception

handling code. Similar findings were also discussed in a prior survey [ECS15].

To improve the quality of exception handling, prior research has reported a slew of anti-patterns on exception handling. These anti-patterns describe the problematic exception handling source code that may exist in the entire life cycle of exceptions, i.e., the propagation of the exception, the flow of the exception and the handling of the exception. Although these anti-patterns are discussed in prior research [SCKB16], the prevalence of these anti-patterns is not studied in-depth.

In this chapter, we investigate the prevalence of exception handling anti-patterns in 16 open-source Java and C# applications and libraries. We find that all of the studied subjects have exception handling anti-patterns detected in their source code. Whereas only five anti-patterns (*Unhandled Exceptions*, *Catch Generic*, *Unreachable Handler*, *Over-catch*, and *Destructive Wrapping*) are prevalently observed, i.e., in median detected in over 20% of the catch blocks or throws statements in the subject systems. We observe that these anti-patterns are often associated with multiple flows of exception, leading to bigger impact and more challenging resolution of such anti-patterns. By further investigation, we find that programming languages (e.g., Java or C#) may have a relationship to the existence of anti-patterns, while we do not observe such relationship with the type of projects (e.g., application or library).

Our results imply that, despite the prior research on exception handling, there is still lacking a deep understanding of the practice of exception handling. More in-depth analyses are needed to ensure the quality and usefulness of exception handling in practice.

The rest of the chapter is organized as follows: Section 4.2 presents our case study methodology. Section 4.3 presents the results of prevalence exception handling anti-patterns. Section 4.4 reveals the amount of exception flows are affect by anti-patterns. Section 4.5 discusses our results. Section 4.6 discusses the threats to the validity of our findings. Finally, Section 4.7 concludes the chapter and discusses potential future research directions based on our early researching findings.

4.2 Methodology

4.2.1 Subject projects

Table 9 depicts the studied subject projects. All subject projects are open-source projects obtained from GitHub. We selected subject projects (see Table 9) by considering their number of stargazers and contributors. These are the same projects discussed in previous chapter, Section 3.2.2, however we include now the information about the number of catch blocks and the number of throws blocks.

Table 9: Overview of the selected subject projects.

	Project	Release Version	Type	# Throws	# of Catch	# Method (K)	KLOC
C#	Glimpse	1.8.6	App.	-	57	1	31
	Google API	v1.15.0	Lib.	-	30	16	628
	OpenRA	release-20160508	App.	-	143	7	125
	ShareX	v11.1.0	App.	-	341	7	177
	SharpDevelop	5.0.0	App.	-	1060	41	923
	SignalR	2.2.1	Lib.	-	105	2	38
	Umbraco-CMS	release-7.5.0	App.	-	615	15	362
Java	Apache ANT	rel/1.9.7	App.	1,622	1139	11	158
	Eclipse JDT Core	I20160803-2000	Lib.	1,686	1655	25	383
	Elasticsearch	v2.4.0	App.	1,782	408	12	108
	Guava	v19.0	Lib.	509	317	10	79
	Hadoop Common	rel/release-2.6.4	Lib.	4,495	1144	14	147
	Hadoop HDFS	rel/release-2.6.4	App.	1,538	586	4	44
	Hadoop MapReduce	rel/release-2.6.4	App.	1,221	367	6	57
	Hadoop YARN	rel/release-2.6.4	Lib.	4,146	1529	29	257
	Spring Framework	v4.3.2.RELEASE	Lib.	5,856	2301	30	349
Total	**-**	**-**	**-**	**22,855**	**11,797**	**230**	**3,866**

4.2.2 Detecting exception handling anti-patterns

We detected all the exception handling anti-patterns presented in Table 1. In particular, we leverage the automated tool described in previous chapter (see 3.2.1). We use Eclipse JDT and .NET Compiler Platform ("Roslyn") to parse the Java and C# source code, respectively. To precisely detect these anti-patterns, we not only parse the try-catch blocks but also analyze the flow of the exceptions. Our exception flow analysis collects the possible exceptions from four different sources: documentation in the code syntax, documentation for third party and system libraries, explicit throw statements, and binding information of exceptions (not available for C#).[1]

4.3 The prevalence of exception handling anti-patterns

Our goal is to put in perspective the existence of exception handling anti-patterns. We collected source code information from a diverse set of subject projects in different programming languages. The knowledge of the prevalence of anti-patterns would help developers improve exception handling practices.

In total, we detected 17 exception handling anti-patterns from the perspective of the catch block,

[1]Source code, binaries and Tableau visualizations with raw data are available online at https://guipadua.github.io/icpc2017.

i.e., whether each catch block contains an anti-pattern. We also detected two exception handling anti-patterns from the perspective of the throws statements. Throws statements are used to indicate the propagation of exceptions explicitly. Since this feature is not available in C#, we only detect throws level anti-patterns in the Java projects.

Table 10: Percentage of affected catch per project per anti-pattern.

	Project	Flow				Handler													# Catch
		Over-catch	**Over-catch and Abort**	**Unhandled Exceptions**	**Unreachable Handler**	**Catch and Do Nothing**	**Catch and Return Null**	**Catch Generic**	**Destructive Wrapping**	**Dummy Handler**	**Ignoring Interrupted Exception**	**Incomplete Implementation**	**Log and Return Null**	**Log and Throw**	**Multi-Line Log**	**Nested Try**	**Relying on getCause()**	**Throw within Finally**	
C#	Glimpse	33.33%	0.00%	12.28%	63.16%	7.02%	7.02%	75.44%	1.75%	21.05%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.51%	0.00%	57
	Google API	40.00%	0.00%	43.33%	60.00%	10.00%	0.00%	56.67%	20.00%	10.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	6.67%	0.00%	30
	OpenRA	23.08%	0.70%	14.69%	58.74%	23.08%	19.58%	76.22%	1.40%	12.59%	0.00%	0.00%	14.69%	0.00%	3.50%	0.00%	0.00%	0.00%	143
	ShareX	65.10%	0.00%	8.50%	24.63%	11.14%	1.47%	90.62%	1.47%	30.79%	0.00%	0.00%	0.59%	0.29%	1.76%	1.17%	0.29%	1.47%	341
	Sharp D.	32.17%	0.09%	40.38%	45.75%	18.30%	10.00%	45.75%	4.15%	13.40%	0.00%	0.38%	3.30%	0.09%	11.79%	8.96%	0.66%	0.66%	1,060
	SignalR	17.14%	0.95%	10.48%	74.29%	6.67%	9.52%	80.00%	0.00%	13.33%	0.00%	0.00%	3.81%	0.95%	4.76%	0.00%	0.00%	0.95%	105
	Umbraco	43.09%	0.00%	16.10%	36.10%	10.57%	6.67%	84.23%	4.72%	17.07%	0.00%	0.16%	1.46%	0.16%	1.79%	1.46%	1.14%	0.00%	615
Java	Apache ANT	31.26%	0.09%	69.80%	6.94%	11.76%	3.34%	17.56%	37.14%	5.09%	2.81%	0.26%	0.53%	0.26%	1.67%	5.79%	0.35%	14.05%	1,139
	E. JDT Core	11.72%	0.24%	69.06%	11.78%	31.24%	11.18%	3.14%	4.71%	7.25%	1.09%	0.06%	0.48%	0.06%	0.06%	2.36%	0.18%	8.22%	1,655
	Elasticsearch	24.26%	0.00%	24.51%	24.51%	10.54%	4.17%	33.82%	31.62%	8.09%	3.43%	0.00%	0.98%	0.00%	1.23%	4.41%	0.98%	3.19%	408
	Guava	19.87%	0.00%	27.44%	37.22%	4.73%	10.09%	26.50%	24.61%	5.05%	7.89%	0.32%	0.95%	0.00%	0.32%	0.95%	6.94%	10.73%	317
	H. Common	25.00%	0.44%	53.41%	16.26%	4.90%	3.85%	18.97%	29.55%	9.70%	4.98%	0.00%	1.66%	0.44%	1.14%	4.02%	1.49%	18.71%	1,144
	H. HDFS	12.46%	0.17%	41.30%	30.55%	3.24%	1.37%	2.22%	34.13%	5.29%	11.43%	0.00%	1.02%	0.68%	1.88%	1.19%	0.85%	4.44%	586
	H. MapReduce	15.80%	0.00%	49.32%	16.08%	3.00%	7.08%	13.35%	41.69%	8.17%	14.99%	0.00%	3.54%	0.54%	1.09%	3.27%	0.82%	30.25%	367
	H. YARN	15.57%	0.39%	43.75%	20.01%	2.55%	6.80%	12.69%	30.80%	10.01%	4.91%	0.13%	1.70%	0.26%	1.64%	2.62%	1.05%	35.71%	1,529
	Spring	28.55%	0.00%	48.46%	25.51%	7.95%	3.00%	29.99%	40.03%	7.82%	0.74%	0.00%	1.56%	0.04%	0.91%	2.17%	1.78%	4.82%	2,301

Table 11: Percentage of affected throws per project per anti-pattern.

	Apache ANT	E. JDT Core	Elastic search	Guava	Hadoop Common	Hadoop HDFS	Hadoop MapReduce	Hadoop YARN	Spring
Throws Kitchen Sink	6.54%	2.19%	0.34%	11.79%	10.23%	10.86%	3.77%	9.62%	8.21%
Throws Generic	1.85%	1.42%	7.13%	7.86%	3.07%	0.59%	4.50%	9.19%	14.05%
# **Throws**	1,622	1,686	1,782	509	4,495	1,538	1,221	4,146	5,856

All anti-patterns are detected at least once in subject projects, while only a small amount of anti-patterns are prevalent. As shown in Tables 10 and 11, all anti-patterns exist in our subject projects. In fact, the least found anti-pattern, *Incomplete Implementation*, can still be found in six projects. This finding implies that prior research indeed captures anti-patterns that correspond to the smell in practice. The existence of all anti-patterns shows the lack of awareness to the importance of quality exception handling code.

On the other hand, we find that only a small number of anti-patterns are prevalent. In particular, only five anti-patterns, i.e., *Unhandled Exceptions*, *Catch Generic*, *Unreachable Handler*, *Over-catch* and *Destructive Wrapping*, are detected in over 20% (40.8%, 31.9%, 28.0%, 24.6%, 22.3%, respectively) of the catch blocks or throws statements in median. On the other hand, all other anti-patterns are rather rare in the source code. Yuan *et al.* [YLZ14] claimed that three exception handling anti-patterns (*Over-catch and Abort*, *Catch and Do Nothing* and *Incomplete Implementation*) could cause catastrophic system failure, while we find that all these three anti-patterns are rarely detected. There are only 12 *Incomplete Implementation* anti-pattern instances detected in all the studied projects. Another surprising finding is that the most widely detected anti-pattern is *Unhandled exceptions*. This anti-pattern has been known as the common root-cause of system crashes [CCHW09], and prior research has proposed techniques to help identify all possible exceptions [RM99, SOH04]. However, our results imply that developers still overlook the importance of this anti-pattern and it may lead to potential crash at system run-time.

4.4 The amount of exception flows

The anti-patterns can be related to a single, multiple, or no exception flow at all (e.g. *Unreachable Handler*). We aim to study the number of flows affected by those anti-patterns. The larger the quantity of flows, the larger the impact of those anti-patterns.

Multiple flows are impacted by each anti-pattern. Table 12 depicts the quantity of affected flows for the flow-based anti-patterns. For *Unhandled Exceptions* and *Unreachable Handler*, 83% (C#) and 67% (Java) of the affected catch blocks have multiple impacted (uncaught) flows, with a maximum of 37 flows. For *Over-catch* and *Over-catch and Abort*, 84% (C#) and 60% (Java) of the affected catch blocks have multiple impacted (over-caught) flows, with a maximum of 43 flows.

4.5 Discussion

In this subsection, we aim to understand the existence of anti-patterns from different perspectives.

Programming languages. The prevalence of exception handling anti-patterns can vary between

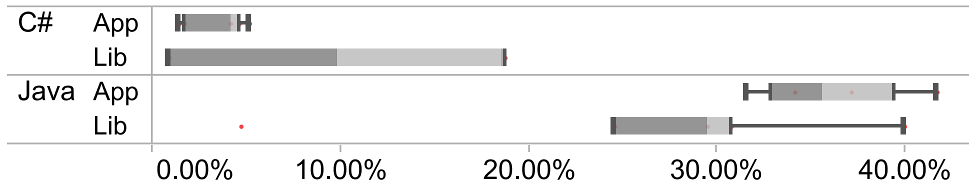
Table 12: Distribution of affected catch blocks according to flow anti-patterns and the quantity of affected flows.

Affected Anti-patterns	**Language**	**Quantity of Flows**					
		1	**2**	**3**	**4**	**5**	**>5**
Unhandled Exceptions and Unreachable Handler	C#	17%	18%	13%	11%	7%	34%
	Java	33%	16%	11%	8%	5%	28%
Over-catch and *Over-catch and Abort*	C#	16%	16%	12%	10%	6%	31%
	Java	40%	17%	12%	7%	6%	19%

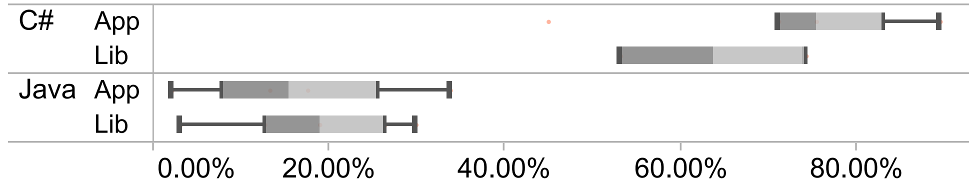
Java and C# (see Table 10). Figure 6 presents examples of anti-patterns that have a large difference in prevalence between Java and C#. The box plots represent the distribution of percentages of catch blocks that contain anti-patterns in each project. For example, the median value of *Destructive Wrapping* in Java (31.6%) is almost 18 times bigger than in C# (1.8%). Another example is *Catch Generic*, in which the minimum value (45.0%, median: 74.3%) in C# is 33% higher than the maximum value (33.8%, median: 17.6%) in Java. The reason of such differences can be the nature of different exception handling strategies in C# and Java. Java forces that certain kinds of exception (i.e. *Checked* exceptions) are handled or explicitly propagated before compilation while C# does not. To support that, popular Java IDEs suggest the exceptions that should be handled. For example, if a developer adds a function call to read a file, the IDE will propose that the non-generic exception *IOException* should be handled or propagated.

Types of projects. Library and application projects may have different exception handling practice, where, intuitively, libraries would propagate exceptions and applications handle exceptions. We examine whether such difference impacts the prevalence of exception handling anti-patterns. Figure 6 presents examples of anti-patterns that have a substantial difference in prevalence between libraries and applications. The differences are not statistically significant (p-value >0.05). We can see that the variance of each distribution is high, which implies that the results may be due to the nature of each project instead of the project type, i.e., library or application.

Generic and non-generic catch blocks. Generic exceptions is an anti-pattern by itself, while some other anti-patterns, e.g., *Dummy Handler*, may be related to Generic catch blocks. We identified that there exists a significant difference (Wilcoxon Rank Sum test, p-value <0.05) between generic and non-generic catch regarding anti-patterns. Generic catch is a sign of developers' lack of knowledge on the possible exception(s), which explains the reason why developer may not know how to handle the exception but only log the exception instead (*Dummy Handler*). On the other hand, since generic catch may cover all possible exceptions from a try block, the chance of having



(a) *Destructive Wrapping*



(b) *Catch Generic*

Figure 6: Examples of differences between Java and C# and between applications and libraries. Differences between Java and C# are significant based on Wilcoxon Rank Sum test (p-value <0.05). Based on the same test, all differences between applications and libraries are not statistically significant.

Unhandled exception anti-pattern is smaller. Yet, such exception handling may mix critical issues with minor issues by only superficial handling strategies (like *Dummy Handler*), which may cause catastrophic failures of the software.

Runtime and non-runtime exceptions. Software is expected to recover more from non-runtime exceptions than runtime exceptions [GJS+15]. We compare anti-patterns detected with runtime and non-runtime exceptions in non-generic catch blocks, since generic exceptions are typically non-runtime. We find significant differences (Wilcoxon Rank Sum test, p-value <0.05) for *Destructive Wrapping, Incomplete Implementation and Throw within Finally*, only in Java projects. In all of those, the percentage of affected catch blocks is lower for runtime exceptions. Java does not force developers to handle runtime exceptions. Therefore, they are handled only if developers well understand the runtime exceptions, leading to fewer anti-patterns.

4.6 Threats to validity

4.6.1 External validity

Our findings may not generalize for other software, other programming languages or commercial software.

4.6.2 Internal validity

Our study may not cover all possible anti-patterns. We selected anti-patterns based on the current research in the subject. Some anti-patterns that either 1) are out of the scope of exception handling (yet still mentioned in related work), 2) require heuristic to detect or 3) are not well explained in details in related work, are not included. Missing necessary documentation may also impact the identification of anti-patterns.

4.6.3 Construct validity

The results in our study are based on catch blocks and throws statements. There may be other ways to measure the exception handling anti-patterns and their prevalence. Some anti-patterns might be mitigated according to the use of newer exception handling features (e.g., multi-catch [Rie11, AARS16]).

4.7 Conclusion

In this chapter, we perform an empirical study using automatically detected 19 exception handling anti-patterns in 16 open-source projects. We find that although all studied projects contain exception handling anti-patterns and every anti-pattern is detected in the source code, there exist only a small number of anti-patterns that are prevalent. These anti-patterns are often associated with multiple exception flows, making them more impactful and more difficult to address. With further investigation on the prevalence of anti-patterns, we find that the choice of programming languages may have a relationship to the introduction of anti-patterns. Our results suggest the need of in-depth study on exception handling practices. In particular, more user studies are required to further understand the choices of exception handling code and the introduction of exception handling anti-patterns. More importantly, future work should consider the impact of such exception handling code to assist in better resolution of exception handling anti-patterns and issues.

Chapter 5

Studying the Relationship between Exception Handling Practices and Post-release Defects

Previous chapters discussed the exception handling practices: the flow characteristics (i.e., Chapter 3) and their anti-patterns (i.e., Chapter 4). However, little is known about the relationship between exception handling practices and software quality. In this chapter, we investigate the relationship between software quality (measured by the chance of having post-release defects) and: (i) exception flow characteristics and (ii) 17 exception handling anti-patterns. We perform a case study on three Java and C# open-source projects. By building statistical models of the chance of post-release defects using traditional software metrics and metrics that are associated with exception handling practice, we study whether exception flow characteristics and exception handling anti-patterns have a statistically significant relationship with post-release defects. We find that exception flow characteristics in Java projects have a significant relationship with post-release defects. In addition, although majority of the exception handing anti-patterns are not significant in the models, there exist anti-patterns that can provide significant explanatory power to the chance of post-release defects. Therefore, development teams should consider allocating more resources to improving their exception handling practices, and avoid the anti-patterns that are found to have a relationship with post-release defects. Our findings also highlight the need for techniques that assist in handling exceptions in the software development practice.

5.1 Introduction

Modern programming languages, such as Java and C#, typically provide exception handling features, such as throw statements and try-catch-finally blocks. These features separate error-handling code from regular source code and are leveraged widely in practice to support software comprehension and maintenance [MSR85, CCHW09].

Having acknowledged the advantages of exception handling features, their suboptimal usage can still cause catastrophic software failures, such as application crashes [YLZ14, KZP+13], or reliability degradation, such as information leakage [Car96, ZC14]. A large portion of systems has suffered from system crashes that were due to exceptions [Cri82]. Additionally, the importance of exception handling source code has been illustrated in prior research and surveys [BGB14, ECS15].

Prior studies aim to understand the practices of exception handling in its different components: exception sources and handling code [SCKB16]. Findings from those empirical studies have advocated the suboptimal use of exception handling features in open source software [NHT16, KLM16, AARS16, BCR+15, BdPS17a]. Moreover, exception handling anti-patterns that are defined by prior research [YLZ14, CCHW09, McC06, BGB14] are observed to be prevalent in open source projects [BdPS17b]. These prior research findings imply the lack of a thorough understanding of the practice of exception handling. If the suboptimal practices do not share a relationship with software quality, our results may provide evidence to explain the findings from prior studies. However, little is known about the existence of such relationship.

Therefore, in this chapter, based on the previous findings of suboptimal exception handling practices (i.e., anti-patterns and flow characteristics), we perform an empirical study of the relationship between exception handling practices and post-release defects (as a proxy to software quality). In particular, our case study is conducted on two open source Java projects (Hadoop and Hibernate) and one open source C# project (Umbraco). Through the case study results, we would like to answer the following two research questions:

RQ1: Do exception flow characteristics contribute to better explaining the chance of post-release defects?

RQ2: Do exception handling anti-patterns contribute to better explaining the chance of post-release defects?

We find that, in some project (e.g., Umbraco), we do not observe any statistically significant relationship between exception flow characteristics and post-release defects. However, in the other two Java projects, the suboptimal practices of exception handling (e.g, the ambiguity of possible exceptions) indeed have a statistically significant relationship with post-release defects. In addition, although majority of the anti-patterns do not have a statistically significant relationship with post-release defects, four anti-patterns are observed to be statistically significant. More importantly,

these anti-patterns may be prevalent ones and may provide large explanatory power to the chance of post-release defects in the studied projects.

Our case study results imply the importance of avoiding suboptimal exception handling practices. Furthermore, although not all anti-patterns are shown to be harmful, developers should at least consider avoiding the ones that are found to have a relationship with post-release defects in this study. Our findings can be used as a guideline for avoiding suboptimal exception handling practices.

The rest of the chapter is organized as follows: Section 5.2 describes the design of our case study. Section 5.3 presents the results of our case study. Section 5.4 discusses the threats to the validity of our findings. Finally, Section 5.5 concludes the chapter and discusses its implications.

5.2 Case Study Design

In this section, we present the design of our case study. We first present our research questions. We then describe the studied systems. Finally, we present our metrics, modeling approach and relevant preliminary results.

5.2.1 Research questions

The general goal of this chapter is to understand whether suboptimal exception handling practices have a relationship with the chance of post-release defects. To achieve the goal of the chapter, in this subsection, we discuss our formulated research questions and their motivation.

As discussed in Section 2, prior studies often expose the suboptimal exception handling practices in two ways. First, they generally quantify the exception handling characteristics. Second, they define particular exception handling anti-patterns. Although prior studies claimed that some quantified exception handling characteristics (e.g., handling exceptions using the generic handling strategy) and exception handling anti-patterns are undesired, practitioners still often suboptimally use exception handling without considering the impact of such inadequate practices. [SCKB16].

On one hand, maybe such undesired exception handling does not impact software quality in practice. On the other hand, lacking statistically rigorous empirical evidence, practitioners may not be aware of such impact, leading to the prevalence of suboptimal exception handling practices (e.g., anti-patterns) in their source code.

Therefore, we formulate two research questions, according to the two ways of unveiling suboptimal exception handling practices by prior research.

RQ1: Do exception handling flow characteristics contribute to better explaining the chance of post-release defects?

RQ2: Do exception handling anti-patterns contribute to better explaining the chance of post-release defects?

We choose to use post-release defects as one widely used indicator of software quality. Since there exist traditional software metrics that are shown to have a statistically significant relationship with software quality, we would like to understand whether the suboptimal exception handling practices provide additional information to complement the traditional metrics in explaining software quality (i.e., post-release defects in this chapter).

5.2.2 Subject projects

Table 13 depicts the overview of the studied subject projects specifically for the case study of this chapter. We consider Java and C# due to their popularity and that they are widely studied in prior research (see Section 2). Moreover, the different approaches of exception handling between Java and C# may further help us understand our study results. To facilitate replication of our work, we opt to study open-source projects that are available on GitHub.

We leverage GitHub filters on the number of contributors (i.e. projects with multiple contributors) and the number of stargazers (i.e. projects with more than ten stargazers), as they are found to be good indicators for selecting engineered software projects [MKCN17]. To narrow down the number of projects, we also prioritize on the projects with higher numbers of stargazers and larger project sizes in terms of lines of code.

After reading the official description of the projects, we investigate the traceability of information in the projects issue tracker. Similar to previous research (see Section 2), the post-release defects should be reasonably straightforward to trace to source code files. From each project, we inspected the release notes of their most recent stable version of the source code at the moment of data collection for analysis. We selected the versions that have had a higher number of post-release updates. In the end, to better understand post-releases defects related to exception handling, the three subject projects and their corresponding releases are chosen also due to (i) the number of files with catch blocks; (ii) the number of files with post-release defects.

5.2.3 Metrics

In order to study the relationship between exception handling practices and post-release defects, we extract metrics based on the analysis of source code, development history of the version control system and issue tracking systems of the subject projects. We extract four categories of metrics for our study.[1]

[1]Detailed metric definition and aggregation rules, data, R notebooks are available online at https://guipadua.github.io/eh-model-defects2018.

Table 13: An overview of the subject projects.

	Umbraco	**Hadoop**	**Hibernate**
Language	C#	Java	Java
Purpose	CMS	Big Data tool	Database ORM
Release Version (tag name)	release-7.6.0	release-2.6.0	5.0.0.Final
Latest Post Release Version (tag name)	release-7.6.12	rel/release-2.6.5	5.0.16
# Files	3174	3698	3488
# SLOC (K)	247	859	271
# Pre Release Changes	1182	2753	11855
# Pre Release Defects	126	673	3038
# Post Release Changes	317	593	672
# Post Release Defects	112	383	499
# Files with Post Release Defects	93	226	356
# Catch	647	5939	1546
# Files with Catch	321	926	478

Post-release defects

We first extract post-release defects of each source code file of the subject projects. We only consider the fixed defects in the issue tracking systems. We use the ID of the defects to identify code changes on the corresponding files that fix such defect. We compare the defect report time and the release date of the subject project to determine whether the defect is a post-release defect or not.

Traditional product metrics

Prior research on defect modeling finds that product metrics such as size (e.g., lines of code) and complexity (e.g., cyclomatic complexity) are good indicators of post-release defects [DLR10]. Therefore, we use *Understand* [Scia] on the release version of the source code of the subject projects to extract traditional product metrics. In particular, we extract all the 39 file level product metrics that are provided by *Understand* for both Java and C#. [Scib].

Traditional process metrics

Process metrics are found to be more powerful in defect modeling than product metrics [MPS08]. We extract traditional process metrics from the development history of the subject projects. In particular, we extract three categories of the traditional process metrics:

- **Change metrics.** We calculate the change metrics based on pre-release changes using the

specific release branch for a given version. For pre-release changes, we used specific pre-release branches, the date range based on the subject release notes and the oldest change associated with the release. We calculate the total number of changes and total code churn as two change metrics.

- **Human factors.** Code ownership is observed to have a relationship with software defects [RD11]. We use the number of unique authors of a file as a proxy for code ownership. We calculate the number of unique authors by checking the associated e-mail address of a change in the development history of a file.

- **Pre-release quality metrics.** Prior research finds that pre-release defects are a good indicator of the chance of post-release defects [MPS08, NBZ06]. Therefore, we extract the number of pre-release defects by following a similar approach to extracting post-release defects that are explained above.

Exception handling metrics

To study exception handling practice, we extract two sets of the exception handling metrics in order to answer the two research questions.

- **Exception flow characteristics metrics.** This set of metrics describes the characteristics of exception flow. As discussed in Section 2, such characteristics often unveil the suboptimal exception handling practices. Tables 14, 15, 16 describes the metrics. Each metric is calculated using its total amount and its average value.

Table 14: Exception handling flow characteristics metrics: part one of three. The symbol †indicates the rows where each metric represents multiple metrics.

Metric	Description	Rationale
Flow Quantity	The distinct number of possible exceptions that arrives in the handler.	The more exceptions there are, the more challenging it is for developers to handle all exceptions [BdPS17a]. Missing handling exceptions is one of the causes of exception handlings defects. [ECS15].
Flow Quantity - Propagated	The distinct number of possible exceptions that are propagated by the handler.	Propagated exceptions need to be handled elsewhere. If they remain uncaught, there could be a risk of system failures [SOH04, BGB14, ECS15].
Flow Quantity - Propagated and Potentially Recoverable	The distinct number of potentially recoverable possible exceptions that are propagated by the handler.	Recoverable exceptions are expected to be handled [GJS+15, .NE]. Leaving recoverable exceptions unhandled might increase the chance of defects since developers and users do not expect they will happen.
Flow Type Prevalence	The average prevalence (i.e., measured among all try blocks of the project) of the flow exception types of a try block.	Although multiple exception types exist in each project, many appear in only one try block [BdPS17a]. However, a rare exception could represent a higher chance of defects since developers might not be familiar with how to handle it.
Flow Sources - Declared	The average number of declaring method(s) per possible exception of a try block.	Although an exception might be traced from different invoked methods [BdPS17a], it might be declared in a unique method. There might be a higher chance of defects if there is a higher number of declared methods.
Flow Sources - Invoked	The average number of invoked method(s) per possible exception of a try block.	Having multiple sources for an exception might increase the chance of defects since it creates ambiguity for developers/testers handling/testing the different possible control flow paths of such exception [BdPS17a].

Table 15: Exception handling flow characteristics metrics: part two of three. The symbol †indicates the rows where each metric represents multiple metrics.

Metric	Description	Rationale
†Flow Sources - Documentation	The percentage of the possible exceptions of a try block found by a given exception documentation source (i.e., throw statements, comments, external documentation, and, for Java, method declaration).	Lacking immediate documentation is one of the challenges of exception handling [CM07, SCKB16, KS14]. Lacking documentation of each different documentation source might increase the chance of defects.
†Flow Handling Strategy	The percentage of the possible exceptions of a try block that is handled with a given strategy (i.e, specific and subsumption).	The subsumption handling strategy introduces harmful uncertainty [SOH04, SCKB16, BdPS17b] and, therefore, could increase the chance of defects. Nevertheless, only a small portion of the exceptions are handled with the specific strategy [BdPS17a]. Such strategy might not reduce the chance of defects.
†Flow Handling Actions	The number and the percentage of possible exceptions of a try block handled with a given action (i.e., 12 different actions [BdPS17a]).	Chapter 3 indicated differences in the prevalence of handling actions [BdPS17a]. Proper recovery actions taken during handling would reduce the chance of defects, meanwhile inappropriate actions could reveal a higher chance of defects.

Table 16: Exception handling flow characteristics metrics: part three of three. The symbol †indicates the rows where each metric represents multiple metrics.

Metric	**Description**	**Rationale**
Try Quantity	The number of try blocks in the file.	Try blocks can affect the normal control flow of the program. Such increase can potentially lead to more defects in the file, as it becomes more complex.
Try Size - LOC	The number of lines of code in the try blocks of the file.	Longer try blocks are more complex and include more code that could potentially go through abnormal situations and have their flow altered due to an exception.
Try Size - SLOC	The number of source lines of code in the try blocks of the file.	The lines of code in a try block might not be source code (e.g., comments). Therefore, we aim to focus on the effective number of source lines of code as an indicator for a higher chance of defects in the case of a high number of lines (i.e., similar to Try Size - LOC).
Invoked Methods	The number of invoked methods in a try block.	Try blocks with more invoked methods can potentially have more possible exceptions and are inherently more complex. These methods also can be affected by an exception event since they might not be executed at runtime. More methods could mean a higher chance of defects.
Try Call Depth	The average relative (i.e. to the handler) call graph depth in which a possible exception was found for this handler.	The large distance between throw and catch makes the exception handling less meaningful and testing and debugging more difficult [SOH04, RS03]. Therefore, a higher distance will likely increase the number of defects of a file.
†Try Scope	Scope in which the try statement was declared: Declaration, Condition, Loop, EH Feature, Other	Nested exception handling constructs are harder to read, test and maintain. [CCHW09, CM07, ECS15]. The try scope can be a possible factor to increase the chance of defects since, for example, a try-block nested in a loop would be harder to understand than a simple declaration.

Table 17: Exception handling anti-patterns metrics. The symbol †indicates the rows where each metric represents multiple metrics.

Metric	Description	Rationale
†Catch Anti-patterns	The number and the percentage of handlers affected by a given anti-pattern (i.e., 17 different anti-patterns on Table 1).	Anti-patterns compromises the robustness of the program and can lead to defects [CCHW09, McC06, KPGA12]. Exception handling anti-patterns are prevalent [BdPS17b] and it may increase the chance of defects.
†Catch Recoverability	The recoverability of the exception type declared in the catch block.	Potentially unrecoverable exceptions are more challenging to handle [GJS+15, .NE]. A higher amount of exception handling for potentially unrecoverable exceptions may be associated with less reliable code.
Catch Quantity	The number of catch blocks in the file.	Catch blocks are only executed during exceptional events. A higher number of catch blocks may be related with more exceptional scenarios of the execution, leading to a higher chance of defects.
Catch Size - LOC	The number of lines of code in the catch blocks of the file.	Longer catch blocks include more code that take measures in the event of an exception. This could increase the chance of bugs since it indicates a higher complexity and bigger size.
Catch Size - SLOC	The number of source lines of code in the catch blocks of the file.	The lines of code in a catch block might not be source code (e.g., comments). Therefore, we aim to focus on the effective number of source lines of code as an indicator for higher chance of defects in the case of a high number of lines.

- **Exception handling anti-pattern metrics.** This set of metrics describes the anti-patterns of exception handling since the anti-patterns are claimed to be harmful to software quality. Table 1 describes all the anti-patterns that are considered in this study. We do not consider the *throws* anti-patterns since they do not apply for C# projects. In particular, each of the 17 catch (i.e., flow and handler) anti-patterns have two metrics that measures (i) the total amount and (ii) the average number of catch blocks that are impacted by the anti-pattern. In order to provide the basic information about exception handling blocks (catch blocks), we also calculate four additional metrics as shown in Table 17.

In order to extract these metrics, we use our tools developed in previous chapters. These tools use Eclipse JDT and .NET Compiler Platform ("Roslyn") to parse Java and C# source code, respectively. To precisely detect anti-patterns, the tools not only parse the try-catch blocks but also analyze the flow of the exceptions. The tools' exception flow analysis collects the possible exceptions from four different sources: documentation in the code syntax, documentation for third party and system libraries, explicit throw statements, and binding information of exceptions (not available for C#).

5.2.4 Model construction

We build logistic regression models to evaluate the explanatory power of the exception handling practices on post-release defects. Regression models require less data than machine learning and it is capable of providing exact understanding for each predictor [Har15]. Similarly to previous studies [SNH15, MKAH16], we consider the explanatory power of the traditional metrics that are empirically known to have a relationship with post-release defects. For that reason, we first build a base model (i.e, *BASE*) with only the traditional software metrics and without the metrics that are associated with exception handling practices. Section 5.2.3 and 5.2.3 details the traditional metrics that are used in the base model. Afterwards, we construct a combined model called *BSFC* by adding software metrics that are associated with quantified exception flow characteristics from prior studies [BdPS17a] into the base model. We also add software metrics that are associated with the exception handling anti-patterns from prior studies [BdPS17b] into the base model to construct a second combined model called *BSAP*. By examining the significance and the explanatory power of the metrics in *BSFC* and *BSAP*, we answer our two research questions, respectively. In the rest of the subsection, we present the detail of our model construction process as illustrated by Figure 7.

MC1: Missing data analysis

After extracting metrics from the data, we might still have missing data. We manually examine the files with missing data. We find that the reasons may due to the cases where the file is not compilable

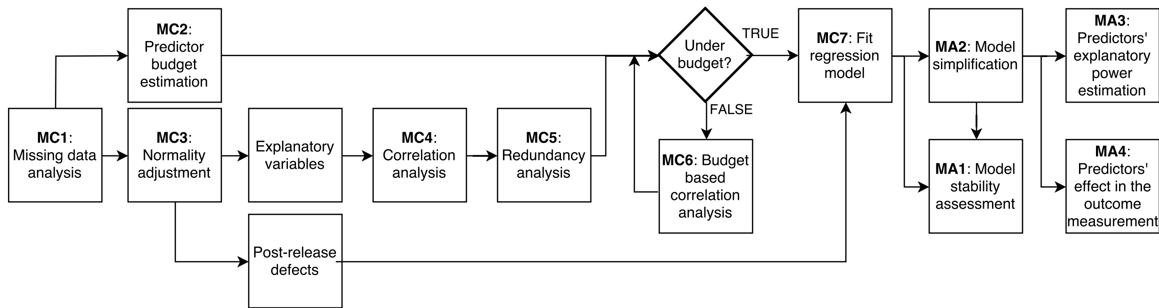


Figure 7: An overview of our modeling approach: model construction and model analysis.

or cases in which the methods of a try block actually doesn't throw any exception (e.g., forgotten try blocks during code evolution). As recommended by statistical modeling researchers [Har15], we discard the files with missing data since it only stands for less than 3% of the entire data.

MC2: Predictor budget estimation

An *overfitted* model is a statistical model that contains more parameters than the possible amount (i.e., budget) that can be justified by the data. Such model will match the training data too closely and might not be useful to understand the explanatory power of its predictors [Har15]. To lower the chances of overfitting, one can use as a reference the amount of, at least, 15 observations per predictor, which is suggested by prior research on statistical modeling [Har15]. Therefore, in our study, each model will have a budget of the number of files divided by 15.

MC3: Normality adjustment

Logistic regression models expect normality in the outcome and in the predictors. Metrics from software engineering data typically do not follow a normal distribution [MKAH16, SBZ12]. For example, post-release defects exist only in a small portion of the files. Therefore, we apply a log transformation: $log_{10}(x + 1)$ to reduce the skew and adequate the data to the logistic regression assumption.

MC4: Correlation analysis

Software metrics can be highly correlated to each other [EEBGR01]. Highly correlated metrics (i.e., $|\rho| > 0.7$) can be clustered and then represented in regression modeling by a single predictor [Har15]. Prior to modeling, we evaluate the correlations among our extracted metrics. We use Spearman pair-wise rank correlation to better adequate to potential lack of normality in the data. We use the *findCorrelation* method from the *Caret* R Package [Kuh17]. Such method automatically removes the metrics among the highly correlated metrics with the highest mean correlation values.

MC5: Redundancy analysis

Besides pair-wise correlations, we can analyze whether one predictor can be explained based on a model composed of all other predictors [Har15]. This step is executed in an iterative manner in which predictors are dropped until no predictors can be predicted with a R^2 or an adjusted R^2 higher than 0.9. We use the *redun* method from the *Hmisc* R Package [Jr17a]. After we perform correlation analysis and redundancy analysis, we have a list of *potential predictors* for modeling.

MC6: Budget based correlation analysis

We evaluate the number of predictors budget of each project and how many potential predictors exist in the metric set. We consider a given project as over budget if the number of potential predictors is higher than the budget. If the number potential predictors is higher than the budget we execute a new correlation analysis. However, at this time, we use the budget as a target in terms of the number of predictors. For example, if the budget is eight, we run the correlation analysis reducing the correlation cutoff and removing the predictors with higher correlation until we only have eight predictors. We use the *findCorrelation* method from the *Caret* R Package [Kuh17]. During the selection of metrics we force the significant metrics from the BASE model to stay in the model using the *varclus* method from the *Hmisc* R Package [Jr17a].

By using this approach we blind ourselves from the outcome, which is the number of post-release defects. Therefore, we eliminate any bias which other outcome-based approaches could cause in the modeling [Har15]. Nevertheless, we aim to still keep the predictors that are different from each other, which could potentially contribute more to the model.

MC7: Fit regression model

Similar to previous work mentioned in Section 2, we use logistic regressions to model the chance of post-release defects of our subject projects. As we have the final list of predictors, we use the method *lrm* from the *RMS* R Package [Jr17b]. We use logistic regression since we aim to *understand* the likelihood of having post-release defects in a given file instead of building a defect prediction tool.

5.2.5 Model analysis

MA1: Model stability assessment

The initial model analysis is to assess the model fit using the Nagelkerke R^2 (provided by the *lrm* method). The Nagelkerke R^2 is an adjusted version of the Cox & Snell R^2 that adjusts the scale of the statistic to cover the full range from 0 to 1 [N+91], and is an adequate measure for evaluating

competing logistic regression models [HJLS13]. The regular R^2 does not apply to logistic regression and deviance explained is inappropriate [Har15].

However, since the model is built using historical data, there is a chance that unseen observations would reduce the validity of the model. Therefore, to validate our model stability, we use bootstrap with 1,000 repetitions with the function *validate* from the *RMS* R package [Jr17b]. From the bootstrap, we obtain an optimism-reduced Nagelkerke R^2. The optimism-reduced Nagelkerke R^2 accounts for noise among the predictors as well as the model stability with different data sample (i.e., overfitting).

MA2: Model simplification

Not all predictors in the model significantly contribute to the model fit. To simplify the model we apply the fast backward predictor selection technique in the fitted model. Such technique is appropriate since it is not biased and we can judge the impact of the model fit after iteratively removing each insignificant predictor. We use the *fastbw* function from the *RMS* R package [Jr17b]. We use Wald χ^2 test of individual predictors and significance level (i.e., p-value) of 0.05 as our stopping rule. With the remaining predictors, we refit the model for analysis and execute again the assessment of the model stability.

MA3: Predictors' explanatory power estimation

We use the Wald χ^2 test to identify the predictors with the highest explanatory power among the significant predictors. A higher Wald χ^2 indicates a higher contribution to the model fit [Har15].

MA4: Predictors' effect in the outcome measurement

Although the previous step can explain the power of each predictor in the model, we cannot measure what would be the impact of each predictor on the model outcome, i.e., the chance of post-release defects. In this step, similarly to previous research [SNH15, SBZ12], we calculate the model outcome by setting all predictors at their mean value. For each significant predictor, we increase its value by 10% while keeping all other significant predictors at their mean values. We measure the differences of the model outcome as the effect of the predictor. We use the *predict* function from the *RMS* R package [Jr17b].

5.2.6 Preliminary results

As a preliminary analysis, we build models using all available files (i.e., with or without exception handling constructs) from the subject projects. In the preliminary analysis, the only exception

Table 18: A summary of the fitted models' construction and analysis.

	Umbraco			Hadoop			Hibernate		
	BASE	**BSFC**	**BSAP**	**BASE**	**BSFC**	**BSAP**	**BASE**	**BSFC**	**BSAP**
# Predictors Budget	15	15	15	59	59	59	29	29	29
# Potential Predictors	12	23	16	10	31	23	18	27	19
Adjusted Correlation Cutoff	0.70	0.24	0.67	0.70	0.70	0.70	0.70	0.70	0.70
Optimism-reduced Nagelkerke R^2 on Simplified Model	0.09	0.09	0.18	0.33	0.37	0.35	0.21	0.28	0.23

handling metrics we use is the number of exception handling constructs, such as try, catch or throws blocks. If such simple metric is not significant in the models, further analysis on exception handling practices is meaningless. As a result, we find that the number of exception handling constructs is indeed significant in all models and not highly correlated with any other metrics (e.g., for Hadoop, the number of try blocks only has a 0.4 $|\rho|$ correlation with the lines of code.)

By knowing the significance of basic metric of exception handling, we decide to focus only on the files with exception handling constructs since our metrics defined in Section 5.2.3 are only meaningful if there exist exception handling constructs in the file.

5.3 Case Study Results and Discussion

In this section, we present the results of our case study according to our research questions. For each question, we discuss the model construction and the model analysis results that lead to our findings.

RQ1: Do exception handling flow characteristics contribute to better explaining the chance of post-release defects?

Exception flow characteristics of Java projects complement traditional metrics in explaining post-release defects.

Table 18 presents the model fits in optimism-reduced Nagelkerke R^2 on Simplified Models. By comparing the model fits of the BASE model of each project and its corresponding BSFC, we find that in both Java projects, the metrics extracts from exception flow characteristics can statistically significantly improve the fit of the BASE model. Nevertheless, such metrics cannot provide statistically significant explanatory power to the BASE model of Umbraco, even though the optimism-reduced Nagelkerke R^2 of the BASE model is only 9%. By closely looking at the model construction, to reach the budget of the model, many metrics in the BSFC of Umbraco are discarded, leading to a

low correlation threshold of 0.24. Therefore, there may exist metrics with higher explanatory power that were discarded. However, without more data to support our analysis, we cannot claim the complementary explanatory power from exception flow metrics in Umbraco.

The prevalence of the flow exception type has a negative relationship on the chance of post-release defects.

The significant metric with highest χ^2 in BSFC model of Hibernate is the prevalence of particular exception types. Table 19 shows the large explanatory power of the metric on the chance of post-release defects. This result shows that a file with very common exception types (i.e., types that appear in a large number of try blocks of the project as a possible exception) have a lower chance of post-release defects, while files with rare exception types have a higher chance of post-release defects. For example, developers of Hibernate may be familiar with how to handle the common *java.sql.SQLException*. But might not be the case for exceptions such as *org.hibernate.procedure. ParameterStrategyException*. This finding implies that developers should carefully handle files with rare exceptions.

The actions in the catch blocks may have a statistically significant relationship with the chance of post-release defects.

In Hibernate, the files with more possible exceptions handled with *Throw Wrap* action (i.e., HB-6) have lower chances of post-release defects (i.e., negative relationship). *Throw Wrap* means that the original exception or its associated information was wrapped into a throw statement in the catch block. Prior research finds that this action is the most prevalent action in Java [BdPS17a] and we find that this action is present in 55% of the catch blocks in Hibernate. Such wrapping may help in better explain the exception and provide more customized exception types to handle. By examining all the catch blocks in Hibernate, we find that *java.sql.SQLException* and *java.lang.Exception* are the two most handled exception types. In particular, most of the wrapping (i.e., 148 out of 205, or 72%) for *java.sql.SQLException* was done by converting into an exception that is easier to understand by developers. Such wrapping may help developers who use Hibernate as an API to better handle its thrown exceptions. For *java.lang.Exception*, 21% (i.e., 36 cases out 174) of the catch blocks re-throw the exception as *HibernateException*, which aims to help developers distinguish the *java.lang.Exception* thrown by Hibernate and the ones thrown by other APIs in order to handle the exception accordingly.

The files with a higher percentage of handlers using the *Log* action in Hadoop have a higher chance of post-release defects (i.e., positive relationship). The *Log* action is an indicator that the exception is not handled, but, instead, the exception is recorded by logging [SCKB16, CM07, CRG+08]. Moreover, for Hadoop, 64% of the logged catch blocks were handled with a generic exception type

(i.e., IOException 40%, Exception 15% and Throwable 9%) leading to a possible ambiguity of properly handling the exception. Therefore, logging the exception is often required to later (i.e., in the case of a runtime event) examine such exception. Prior research also finds that more logs may indicate that developers have uncertainties about the source code, leading to a positive relationship with the post-release defects [SNH15].

The files with a higher percentage of handlers using the *Method* action in Hadoop have a higher chance of post-release defects (i.e., positive relationship). The *Method* action is when other methods are called in the catch block [SCKB16, CM07]. Invoking other methods often indicates a more complex handling of exceptions. In particular, we find that 13.19% of the catch blocks with the *Method* action handle *com.google.protobuf.ServiceException*. *protobuf* is an external library for data serialization. Developers may face more post-release defects when dealing with data serialization in Hadoop. Other popular methods include *getMessage*, and *println*. Both of them are special cases of the *Log* action that is also found to have a positive relationship with post-release defects.

The characteristics of try blocks may have a statistically significant relationship with the chance of post-release defects.

The average number of invoked methods per possible exception in the try blocks of the file (HB-7) has a positive relationship with the chance of post-release defects in Hibernate. We find that this metric was correlated (i.e.,$|\rho > 0.8|$) with the average number of declaring methods per possible exception. In other words, the files with possible exceptions that are originated from multiple different sources have a higher chance of defects (i.e., positive relationship). Prior research has claimed that an exception that has multiple distinct sources may have an ambiguous meaning when thrown [BdPS17a]. Handling such exception is more challenging and requires a better understanding of the source code by developers.

The average percentage of propagated possible exceptions has a positive relationship with the chance of post-release defects in Hadoop. A large number of possible exceptions may increase the challenge of handling them properly within a file. If a large portion of such exceptions is propagated, it means that the file does not handle the exceptions and the responsibility is transferred to the callers of the methods of the file. Propagating exceptions is an easy way to transfer the risk of handling an exception instead of taking action to recover from the exception. However, the exceptions can still occur and the methods of the file might not work properly since the abnormal behavior was not dealt properly. We consider this chain reaction may be the reason for such positive relationship.

The scope in which the try statement was declared may also have a relationship with the chance of post-release defects. HA-8 is the metric that measures the number of try blocks inside another block that is not declaration, condition, loop or exception handling features. By examining Hadoop's source code, we find that try statements are often declared inside a *SynchronizedStatement* to ensure

the correctness of the exclusive access to an object's state. For example, in Hadoop HDFS class *DatanodeManager*, a method *handleHeartbeat* leverages a try-catch block to access a data node object in a synchronized manner. The higher chance of post-release defects may due to the complexity of the *SynchronizedStatement*.

RQ2: Do exception handling anti-patterns contribute to better explaining the chance of post-release defects?

> *Exception handling anti-patterns complement traditional metrics in explaining post-release defects. However, the majority of the anti-patterns do not provide statistically significant explanatory power to post-release defects.*

We find that, in all three studied projects, at least one anti-pattern is significant in the BSAP models, providing additional explanatory power to the BASE models. In particular, Umbraco has the highest improvement in model fit when adding exception handling anti-pattern related metrics to the BASE model. However, the majority of the exception handling anti-patterns are not statistically significant in explaining post-release defects.

> *The size of the exceptional handling blocks (catch blocks) have a positive relationship with the chance of post-release defects.*

Similar to the findings of our preliminary analysis, the average number of source lines of code in the files' exception handling blocks has a relationship with the chance of post-release defects. This means that if the file has larger exception handling blocks on average, there is a higher chance of defects. Intuitively, this may be due to the correlation between the size of the catch blocks and total lines of code. However, surprisingly we find that the size of exception handling blocks is not highly correlated with other file size metrics. Therefore, the size of the exception handling blocks brings unique information to explain the chance of post-release defects.

> *Some exception handling anti-patterns may have a positive relationship with the chance of post-release defects.*

The percentage of catch blocks affected by the *Dummy Handler* anti-pattern has a positive relationship with the chance of post-release defects in both Umbraco and Hibernate. The *Dummy Handler* anti-pattern indicates that the catch block was superficially handled and might not be really effective in terms of taking care of the exception. In Java, the compiler forces the developers to catch checked exceptions and therefore *Dummy Handler* is often used by developers to make the code compilable [CCHW09, BdPS17b]. However, C# does not force developers to handle exceptions. When there exists a *Dummy Handler*, it may mean that developers intentionally leave the exception

caught by not handled properly, which may lead to severe issues at run-time and also post-release defects.

The total amount of *Generic Catch* anti-pattern has a positive relationship with the chance of post-release defects in Umbraco. The metric has higher explanatory power than the traditional size and complexity metric of the base model (i.e., χ^2 of 14.82 vs 10.01, see Table 19). Prior study finds that this anti-pattern is prevalent in practice [BdPS17b]. It is indeed convenient that developers can use a generic catch block to handle all exceptions. However, exceptions caught by such blocks cannot be properly recovered without the knowledge of the exact type of the exception. Moreover, our results imply the harmfulness of this anti-patterns. Developers should consider avoiding using *Generic Catch* in practice.

The percentage of catch blocks affected by *Ignoring Interrupted Exception* has a positive relationship with the chance of post-release defects in Hadoop. This anti-pattern is related to the Java exception called *InterruptedException*, which is used on concurrent programming with threads. Due to the complex programming feature that is associated with this exception, ignoring the exception is considered an anti-pattern [McC06]. Especially for Hadoop, a platform where concurrency is a major feature of the software, ignoring the exception may be even more harmful. The special context of Hadoop and the nature of the anti-pattern may explain the positive relationship between this anti-pattern and the chance of post-release defects.

The total number of catch blocks affected by *Log and Throw* has a positive relationship with the chance of post-release defects in Hadoop. The *Log and Throw* anti-pattern has been advocated to be harmful [McC06]. Log and throw in a file can make harder for developers to understand where an exception comes from. This anti-pattern could affect software operation since repeated exceptions would show in the logs. This anti-pattern could also affect debugging by preventing developers to find the errors. Although this anti-pattern is not prevalent in practice [BdPS17b] and it was found to have a small effect to the chance of post-release defects (see Table 19), practitioners should still avoid such a suboptimal practice.

5.4 Threats to Validity

In this section, we discuss the threats to the validity of our findings.

5.4.1 External validity

Our study is based on a set of open-source Java and C# projects from GitHub. Our findings may not generalize to other projects, languages or commercial systems. Replicating our study on other subjects may address this threat and further understand the state-of-the-practice of exception

handling.

5.4.2 Internal validity

We aim to include all possible sources of information in our automated exception flow analysis. However, our analysis may still miss possible exceptions, if there is a lack of documentation or the source code is not compilable. Also, the documentation of the exception may be incorrect or outdated. In our analysis, we trust the content of documentation. Therefore, we cannot claim that our analysis fully recovers all possible exceptions nor that the recovered information is impeccable. Further studies may perform deeper analysis on the quality of exception handling documentation to address this threat.

Our study of the relationship between exception handling practice and post-release defects cannot claim causal effects. We do not aim to conduct impact studies in this chapter. The explanatory power of our exception handling metrics on post-release defects does not indicate that exception handling cause defects. Instead, it indicates the possibility of a relationship that should be studied in depth through further studies.

There is room for improvement of the model fit in our statistical models. The model fit may be further improved by adding more predictors to the models in our two research questions. However, this is expected and should not impact the conclusions, i.e., the found relationship between exception handling practices and post-release defects.

5.4.3 Construct validity

Our study may not cover all possible handling actions. We selected actions based on the previous research in the subject [YLZ14, SCKB16, CM07, CRG+08, ZHF+15]. Some actions are not included in our study if they are either 1) require heuristic to detect, or 2) are not well explained in details in related work.

Our possible exception identification approach is based on a call graph approximation from static code analysis. We may still miss possible exceptions due to under-estimation for polymorphism or unresolved method overload. To complement our study, dynamic analysis on the exception flow may be carried out to understand the system exceptions during run-time.

We leveraged a list of software metrics to measure exception handling practices. However, there may exist other aspects of exception handling that we do not measure. Adding more metrics may provide a further understanding of its relationship with post-release defects. In addition, this chapter only focuses on post-release defects as one aspect of software quality. There exist other aspects of software quality other than post-release defects. For example, exception handling might have

relationship with software maintainability or software understandability. Similar to previous research [SBV+17, PAK+14, KDPG09, KG08], one may consider extending our study by modeling other aspects of software quality.

We leverage an automated approach to remove predictors in order to keep the number of predictors under modeling budget. Another approach to resolving this issue is using expert knowledge [Har15]. Expert knowledge would indicate which predictor should not be considered. We do not opt to leverage expert knowledge since we want to avoid subjective bias in the results. However, the approach of using expert knowledge can be leveraged if closely working with practitioners on this empirical study. Such a study is already in our future plan.

5.5 Conclusion

Exception handling is an important feature in modern programming languages. Prior studies unveil the suboptimal usage of exception handling features in practice and propose exception handling anti-patterns. In this chapter, we study whether the exception handling practices, including the characteristics of exception flow and the exception handling anti-patterns, have a statistically significant relationship with post-release defects. We find that exception flow characteristics in Java projects have a significant explanatory power when complementing traditional software metrics in modelling post-release defect. Such results imply the importance of properly handling exceptions. In addition, although majority of the exception handling anti-patterns are not significant in explaining post-release defects, there exist some anti-patterns that indeed have a positive relationship with post-release defects. Developers should try to avoid such anti-patterns in practice.

In particular, the contributions of this chapter are:

1. We empirically studies the relationship between exception handling practice and the chance of post-release defects.
2. Our results provide guidelines to practitioners for improving their exception handling practices.

This chapter highlights the importance of avoiding suboptimal exception handling practices and advocates the need for techniques that can improve exception handling in software development practice.

Table 19: Significant metrics in the final models with Wald χ^2 and effect values. Effect is measured by setting a metric to 110% of its mean value, while the other metrics are kept at their mean values. A positive impact (i.e., direction $\nearrow$) means that higher values of the metric, higher chance of bugs.

Project	**ID**	**Metric(s)**	**Direction**	**BASE**	**BSFC**		**BSAP**	
				χ^2	χ^2	**Effect**	χ^2	**Effect**
Umbraco	UM-1	Size and Complexity	$\nearrow$	10.01	10.01	6.9%		
	UM-2	Catch Anti-patterns (Dummy Handler)	$\nearrow$				7.58	2.9%
	UM-3	Catch Anti-patterns (Generic Catch), Catch Recoverability, Catch Quantity, Catch Size (LOC and SLOC)	$\nearrow$				14.82	10.5%
Hadoop	HA-1	Changes and Human Factors	$\nearrow$	101.09	102.74	7.0%	108.4	7.0%
	HA-2	Size and Complexity	$\nearrow$	12.31	7.94	5.7%	10.77	7.8%
	HA-3	Complexity	$\searrow$	8.99			4.7	-5.5%
	HA-4	Complexity	$\nearrow$	6.72				
	HA-5	Catch Anti-patterns (Ignoring Interrupted Exception)	$\nearrow$				12.79	1.4%
	HA-6	Catch Anti-patterns (Log and Throw)	$\nearrow$				4.44	0.3%
	HA-7	Catch Recoverability, Catch Quantity, Catch Size (LOC and SLOC)	$\nearrow$				6.98	2.1%
	HA-8	Try Scope (Other)	$\nearrow$		8.97	0.5%		
	HA-9	Flow Handling Actions (Log)	$\nearrow$		14.78	2.9%		
	HA-10	Flow Handling Actions (Method)	$\nearrow$		4.64	2.1%		
	HA-11	Flow Quantity - Propagated	$\nearrow$		12.51	5.1%		
	HA-12	Flow Handling Strategy (Specific)	$\nearrow$		15.82	7.5%		
Hibernate	HB-1	Changes and Human Factors	$\nearrow$	6.62	5.66	5.1%	7.82	5.3%
	HB-2	Size	$\nearrow$	16.5	13.83	5.9%	13.65	5.2%
	HB-3	Documentation	$\searrow$	14.3	15.62	-7.9%	12.61	-6.4%
	HB-4	Catch Size - SLOC	$\nearrow$				6.52	3.8%
	HB-5	Catch Anti-patterns (Dummy Handler)	$\nearrow$				4.69	0.8%
	HB-6	Flow Handling Actions (Throw Wrap)	$\searrow$		5.84	-2.9%		
	HB-7	Flow Sources (Invoked and Declared)	$\nearrow$		6.21	7.6%		
	HB-8	Flow Type Prevalence	$\searrow$		19.85	-4.6%		

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Exception handling is an important feature in modern programming languages. Prior studies unveil the suboptimal usage of exception handling features in practice and propose exception handling anti-patterns.

In Chapter 3, we revisit the practice of exception handling in 16 open-source software in Java and C#. We confirm that there exist suboptimal manners of exception handling, more importantly, we highlight the opportunities of performing source code analysis to recover exception flows to help practitioners tackle various complex issues of handling exceptions.

In Chapter 4, we perform an empirical study using automatically detected 19 exception handling anti-patterns in 16 open-source projects. We find that although all studied projects contain exception handling anti-patterns and every anti-pattern is detected in the source code, there exist only a small number of anti-patterns that are prevalent. These anti-patterns are often associated with multiple exception flows, making them more impactful and more difficult to address. With further investigation on the prevalence of anti-patterns, we find that the choice of programming languages may have a relationship to the introduction of anti-patterns.

In Chapter 5, we study whether the exception handling practices, including the characteristics of exception flow and the exception handling anti-patterns, have a statistically significant relationship with post-release defects. We find that exception flow characteristics in Java projects have a significant explanatory power when complementing traditional software metrics in modeling post-release defects. Such results imply the importance of properly handling exceptions. In addition, although the majority of the exception handling anti-patterns are not significant in explaining post-release defects, there exist some anti-patterns that indeed have a positive relationship with post-release

defects. Developers should try to avoid such anti-patterns in practice.

In particular, the contributions of our thesis are:

1. We design automated tools that recovers exception flows from both Java and C#.
2. We present empirical evidence to illustrate the challenges and complexity of exception handling in open-source systems.
3. We present empirical evidence of the prevalence of exception handling anti-patterns.
4. Our exception flow and anti-patterns analysis, as an automated tool, can already provide valuable information to assist developers better understand and make exception handling decisions.
5. Our thesis is the first work that empirically studies the relationship between exception handling practice and the chance of post-release defects.
6. Our results provide guidelines to practitioners for improving their exception handling practices.

Finally, this thesis highlights the opportunities and urgency of providing automated tooling to help developers make exception handling decisions during the development of quality and reliable software systems.

6.2 Future Work

Our findings could lead us to envision that more systematic techniques can assist software practitioners in improving the handling of software exceptions. Future work goals would be to improve the practice of handling software exceptions, leading to a better software quality and reliability. We want to make software more reliable through understanding and aiding software engineers when coping with the complexity of handling unexpected situations. This goal can be broken down into different objectives and questions that affect developers, testers, and operators. Here, we suggest future work on the objective of assisting software developers in designing and maintaining exception handling.

What are the major concerns of software developers and maintainers when handling exceptions?

There exists a need for a research effort to identify the best-applied practices of exception handling. For example, a junior developer would not understand which information is required to handle an exception adequately. Future research could conduct interviews (e.g., firehouse interviews) with practitioners from both open-source and commercial software projects. The interview answers will shed light on the developers' decisions during their exception handling activities.

Can automated just-in-time suggestions to support developer when handling exceptions be provided?

Based on current findings and other future user studies, handling exceptions might still be challenging since developers may favor concrete recommendations for handling exceptions. Future research could mine software repositories and model the exception handling decisions using. For example, future research can model whether an exception should be handled and what are the proper actions that are needed to do so.

Bibliography

[AARS16] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. How developers use exception handling in Java? In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 516–519, New York, New York, USA, 2016. ACM Press.

[BCR+15] Rodrigo Bonifacio, Fausto Carvalho, Guilherme N. Ramos, Uira Kulesza, and Roberta Coelho. The use of C++ exception handling constructs: A comprehensive study. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30. IEEE, sep 2015.

[BdPS17a] Guilherme B. de Pádua and Weiyi Shang. Revisiting exception handling practices with exception flow analysis. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 11–20, Sept 2017.

[BdPS17b] Guilherme B. de Pádua and Weiyi Shang. Studying the prevalence of exception handling anti-patterns. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 328–331, Piscataway, NJ, USA, 2017. IEEE Press.

[BGB14] Eiji Adachi Barbosa, Alessandro Garcia, and Simone Diniz Junqueira Barbosa. Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *2014 Brazilian Symposium on Software Engineering*, pages 11–20. IEEE, sep 2014.

[CAG+17] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. Exception handling bug hazards in android. *Empirical Softw. Engg.*, 22(3):1264–1304, June 2017.

[Car96] Tom Cargill. *C++ Gems.* SIGS Publications, Inc., New York, NY, USA, 1996.

[CBA+14] Nelio Cacho, Eiji Adachi Barbosa, Juliana Araujo, Frederico Pranto, Alessandro Garcia, Thiago Cesar, Eliezio Soares, Arthur Cassio, Thomas Filipe, and Israel Garcia. How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C#

Applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, number 1, pages 31–40. IEEE, sep 2014.

[CCF+14] Nelio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel Garcia, Eiji Adachi Barbosa, and Alessandro Garcia. Trading robustness for maintainability: an empirical study of evolving c# programs. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, (iii):584–595, 2014.

[CCHW09] Chien-Tsun Chen, Yu Chin Cheng, Chin-Yun Hsieh, and I-Lang Wu. Exception handling refactorings: Directed by goals and driven by bug fixing. *Journal of Systems and Software*, 82(2):333–345, feb 2009.

[CM07] Bruno Cabral and Paulo Marques. Exception Handling: A Field Study in Java and .NET. In *ECOOP 2007 – Object-Oriented Programming*, volume 4609, pages 151–175, Berlin, Heidelberg, 2007.

[CM11] Bruno Cabral and Paulo Marques. A transactional model for automatic exception handling. *Computer Languages, Systems and Structures*, 37(1):43–61, 2011.

[CRG+08] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, volume 5142 LNCS, pages 207–234. 2008.

[Cri82] F Cristian. Exception Handling and Software Fault Tolerance. *Computers, IEEE Transactions on*, C-31(6):531–540, 1982.

[CSM07] Bruno Cabral, P. Sacramento, and Paulo Marques. Hidden truth behind .NET's exception handling today. *Software, IET*, 2(1):233–250, 2007.

[DLR10] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, May 2010.

[ECS15] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software*, 106:82–101, aug 2015.

[EEBGR01] Kalhed El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650, July 2001.

[GC11] Israel Garcia and Nélio Cacho. eFlowMining: An Exception-Flow Analysis Tool for .NET Applications. In *2011 Fifth Latin-American Symposium on Dependable Computing Workshops*, number i, pages 1–8. IEEE, apr 2011.

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. Chapter 11. exceptions - java se specification, feb 2015.

[Har15] Frank E. Harrell ,. *Regression Modeling Strategies*, volume 64 of *Springer Series in Statistics*. Springer International Publishing, Cham, 2015.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[HJLS13] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[JCYC04] Jang Wu Jo, Byeong Mo Chang, Kwangkeun Yi, and Kwang Moo Choe. An uncaught exception analysis for Java. *Journal of Systems and Software*, 72(1):59–69, 2004.

[JGHK13] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. Mining the relationship between anti-patterns dependencies and fault-proneness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 351–360. IEEE, 2013.

[Jr17a] Frank E Harrell Jr. Hmisc: Harrell Miscellaneous, 2017.

[Jr17b] Frank E Harrell Jr. rms: Regression Modeling Strategies, 2017.

[KDPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 75–84, Washington, DC, USA, 2009. IEEE Computer Society.

[KG08] Foutse Khomh and Yann-Gael Gueheneuce. Do design patterns impact software quality positively? In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pages 274–278, Washington, DC, USA, 2008. IEEE Computer Society.

[KLM16] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. Examining programmer practices for locally handling exceptions. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 484–487, New York, New York, USA, 2016. ACM Press.

[KPGA12] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.

[KS14] Maria Kechagia and Diomidis Spinellis. Undocumented and unchecked: exceptions that spell trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 312–315, New York, New York, USA, 2014. ACM Press.

[KSS17] Maria Kechagia, Tushar Sharma, and Diomidis Spinellis. Towards a context dependent java exceptions hierarchy. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 347–349, Piscataway, NJ, USA, 2017. IEEE Press.

[Kuh17] Max Kuhn. caret: Classification and Regression Training, 2017.

[KZP+13] Sunghun Kim, Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, and Shivkumar Shivaji. Predicting method crashes with bytecode operations. In *Proceedings of the 6th India Software Engineering Conference*, ISEC '13, pages 3–12, New York, NY, USA, 2013. ACM.

[McC06] Tim McCune. Exception handling antipatterns, apr 2006.

[MKAH16] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Softw. Engg.*, 21(5):2146–2189, October 2016.

[MKCN17] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating GitHub for engineered software projects. *Empirical Software Engineering*, pages 1–35, 2017.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[MSR85] P. M. Melliar-Smith and B. Randell. Software Reliability: The Role of Programmed Exception Handling. *Reliable Computer Systems*, pages 143–153, 1985.

[MT97] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. *ECOOP'97 — Object-Oriented Programming*, 1241:85–103, 1997.

[N+91] Nico JD Nagelkerke et al. A note on a general definition of the coefficient of determination. *Biometrika*, 78(3):691–692, 1991.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[.NE] Handling and Throwing Exceptions - .NET Framework Documentation.

[NHT16] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in Java projects. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 500–503, New York, New York, USA, 2016. ACM Press.

[OBS+18] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software*, 136:1 – 18, 2018.

[OCC+17] Haidar Osman, Andrei Chiş, Claudio Corrodi, Mohammad Ghafari, and Oscar Nierstrasz. Exception evolution in long-lived java systems. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 302–311, Piscataway, NJ, USA, 2017. IEEE Press.

[PAK+14] Francis Palma, Le An, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Investigating the change-proneness of service patterns and antipatterns. In *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on*, pages 1–8. IEEE, 2014.

[RD11] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[Rie11] Manfred Riem. Working with java se 7 exception changes, sep 2011.

[RM99] Martin P Robillard and Gail C Murphy. Analyzing Exception Flow in Java Programs. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symp. on Foundations of Software Engineering*, pages 322–337, 1999.

[RM03] Martin P Robillard and Gail C Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, 2003.

[RS03] Darell Reimer and Harini Srinivasan. Analyzing exception usage in large java applications. In *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, pages 10–18, 2003.

[SBV+17] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 417–427, Piscataway, NJ, USA, 2017. IEEE Press.

[SBZ12] Emad Shihab, Christian Bird, and Thomas Zimmermann. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 301–310, New York, NY, USA, 2012. ACM.

[Scia] Scitools.com. Understand: Visualize your code.

[Scib] Scitools.com. What metrics does understand have?

[SCKB16] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. Understanding the exception handling strategies of Java libraries. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 212–222, 2016.

[SNH15] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, feb 2015.

[SOH04] S. Sinha, A. Orso, and M.J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit flow control. In *Proceedings. 26th International Conference on Software Engineering*, pages 336–345, 2004.

[TKZ+13] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 270–279, Washington, DC, USA, 2013. IEEE Computer Society.

[TX09] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 496–506, Washington, DC, USA, 2009. IEEE Computer Society.

[WW64] Frank Wilcoxon and Roberta A Wilcox. *Some rapid approximate statistical procedures.* Lederle Laboratories, 1964.

[YLZ14] Ding Yuan, You Luo, and Xin Zhuang. Simple Testing Can Prevent Most Critical Failures. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[ZC14] Benwen Zhang and James Clause. Lightweight automated detection of unsafe information leakage via exceptions. *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 327–338, 2014.

[ZHF+15] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to Log: Helping Developers Make Informed Logging Decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 415–425. IEEE, may 2015.

[ZK09] Lingli Zhang and Chandra Krintz. As-if-serial exception handling semantics for Java futures. *Sci. of Computer Programming*, 74:314–332, 2009.

[ZKZH14] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. An empirical study of the effect of file editing patterns on software quality. *J. Softw. Evol. Process*, 26(11):996–1029, November 2014.