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

Guilherme B. de Pádua and Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
{g_bicalh,shang}@encs.concordia.ca

ABSTRACT

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 revealed the suboptimal practices of the exception handling flows and the prevalence of their anti-patterns. However, little is known about the relationship between exception handling practices and software quality. In this work, we investigate the relationship between software quality (measured by the probability 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 probability 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 the 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 probability 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.

CCS CONCEPTS

• **Software and its engineering → Software reliability**; **Error handling and recovery**; **Maintaining software**; *Software defect analysis*;

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196435

KEYWORDS

Exception Handling,Software Quality,Empirical Software Engineering

ACM Reference Format:
Guilherme B. de Pádua and Weiyi Shang. 2018. Studying the Relationship between Exception Handling Practices and Post-release Defects. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196398.3196435

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 [13, 37].

Having acknowledged the advantages of exception handling features, their suboptimal usage can still cause catastrophic software failures, such as application crashes [33, 56], or reliability degradation, such as information leakage [12, 57]. A large portion of systems have suffered from system crashes that were due to exceptions [16]. Additionally, the importance of exception handling source code has been illustrated in prior research and surveys [5, 18].

Prior studies aim to understand the practices of exception handling in its different components: exception sources and handling code [50]. Findings from those empirical studies have advocated the suboptimal use of exception handling features in open-source software [2, 3, 6, 31, 42]. Moreover, exception handling anti-patterns that are defined by prior research [5, 13, 35, 56] are observed to be prevalent in open-source projects [4]. 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 paper, 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 handling flow characteristics contribute to a better explanation of the probability of post-release defects?

RQ2: Do exception handling anti-patterns contribute to a better explanation of the probability of post-release defects?

We find that, in some projects (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 probability 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.

In summary, the contributions of our paper are:

(1) Our paper is the first work that empirically studies the relationship between exception handling practice and the probability of post-release defects.
(2) Our results provide guidelines to practitioners for improving their exception handling practices.

The rest of the paper is organized as follows: Section 2 discusses related prior research of this work. Section 3 describes the design of our case study. Section 4 presents the results of our case study. Section 5 discusses the threats to the validity of our findings. Finally, Section 6 concludes the paper and discusses its implications.

2 RELATED WORK

In this section, we present the prior research that is related to this paper. In particular, we present the prior research on exception handling practices and the prior research on software defect modeling.

2.1 Exception handling practices

Prior research conducted empirical studies with source code, development history, issue tracking systems and developer surveys to understand the exception handling practice. All of the studies provide empirical evidence that unveils the existence of suboptimal exception handling practices.

Empirical studies are conducted in order to understand the exception handling practices in general. Cabral and Marques [7] studied exception handling practices from 32 projects in both Java and .Net. The study results unveil suboptimal practice of exception handling. Sena et al. [50] investigated sampled exception flows from 656 Java libraries for flow characteristics, handler actions, and handler strategies. A recent study by B. de Padua and Shang [3] revisits exception flow analysis by looking into a higher number of flows per system. The authors included C# .NET systems and considered factors that can impact exception handling practices such as the differences between applications and libraries. Osman et al. [44] differentiates applications and libraries in terms of the usage of exception handling in an evolutionary study of Java systems. Cacho et al. [10, 11] 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. Oliveira et al. [43] 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.

Some empirical studies focus on one or some special aspects of exceptional handling. Jo et al. [26] focus on uncaught checked exceptions in Java projects. The authors proposed an inter-procedural analysis based on set-based framework without using declared exceptions. Coelho et al. [15] assessed exception handling strategy with exception flows from Aspect-oriented systems and object-oriented systems. The authors evaluate the number of *uncaught* exceptions, exceptions caught by *subsumption*, and exceptions caught with *specialized* handlers. Some studies reveal that developers consider exception handling hard to learn and to use and tend to avoid it or misuse it [2, 31, 42]. Bonifacio et al. [6] also surveyed C++ developers encountering revelations of educational issues.

Undesired practices, especially defined anti-patterns, or rules are proposed as indicators of suboptimal exception handling practices [5, 13, 35, 50, 56]. However, such anti-patterns are still found to be prevalent [4].

Sinha et al. [53] leveraged exception flow analyses to study the existence of 11 anti-patterns in four Java systems. Other research [5, 13, 18] classified exception-handling related defects by mining software issue tracking. Thummalapenta and Xie [55] presented a rule-based approach and detected 160 defects, including 87 new defects not previously known, from 294 real exception-handling rules in five applications. Coelho et al. [14] 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 signaled by third-party code.

Prior research also highlights the lack of documentation of exceptions. Kechagia and Spinellis [30] found that 69% of the methods had undocumented exceptions and 19% of crashes could have been caused by insufficient documentation. Sena et al. [50]'s findings confirm that API runtime exceptions are poorly documented. Cabral and Marques [9] identify that infrastructure (20%) and libraries (15%) have better exception handling documentation when compared to applications (2%).

Besides understanding exception handling practices, other work revealed the opportunities of leveraging various analysis to combine information from different sources to understand and assist in exception handling flows and practices [4, 8, 20, 29, 47, 53, 59, 60]. 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 paper, 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 probability of having post-release defects, as one indicator of software quality.

2.2 Software quality and defect modeling

there exists 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. [19] revealed that size is a common confounding factor for the previously defined object-oriented metrics. In a different work, D'Ambros et al. [17] presented a benchmark for defect prediction comparison in terms of the the explanatory and the 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 [23] 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. [52] consider branching activities; Zhang et al. [58] examine editing patterns, Shang et al. [51] investigate logging characteristics and McIntosh et al. [36] study code reviews.

Moreover, researchers investigated the use of programming patterns and anti-patterns and their impact on software quality. Khomh et al. [32] and Taba et al. [54] 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. [32] and Taba et al. [54]: used logistic regression; tested which anti-patterns impact more and showed that size alone cannot explain defective classes. Moreover, Jaafar et al. [25] demonstrated that dependencies to classes with anti-patterns increase the probability of post-release defects.

To the best of our knowledge, this paper 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.

3 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.

3.1 Research questions

The general goal of this paper is to understand whether suboptimal exception handling practices have a relationship with the probability of post-release defects. To achieve the goal of the paper, 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. [50].

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 a better explanation of the probability of post-release defects?

RQ2: Do exception handling anti-patterns contribute to a better explanation of the probability of post-release defects?

We choose to use post-release defects as one widely used indicator of software quality. Since traditional software metrics exist which have been 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 paper).

3.2 Subject projects

Table 1 depicts the overview of the studied subject projects. 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 have been found to be good indicators for selecting engineered software projects [39]. 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.

Table 1: An overview of the subject projects.

Project Characteristic	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

3.3 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]

3.3.1 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.

3.3.2 Traditional product metrics. Prior research on defect modeling found that product metrics such as size (e.g., lines of code) and complexity (e.g., cyclomatic complexity) are good indicators of post-release defects [17]. Therefore, we use *Understand* [48] 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#. [49].

3.3.3 Traditional process metrics. Process metrics are found to be more powerful in defect modeling than product metrics [38]. 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 [45]. 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 probability of post-release defects [38, 40]. Therefore, we extract the number of pre-release defects by following a similar approach to extracting post-release defects that are explained above.

3.3.4 Exception handling metrics. To study exception handling practices, we use two sets of the exception handling metrics that we revisited in our two previous studies [3, 4]. With those metrics, we can 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. Table 2 describes the metrics and the rationale of including such metrics in the modeling. Each metric is calculated using its total amount and its average value.
- **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. Our previous study [4] 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 anti-patterns has 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 3.

In order to extract these metrics, we use our exception flow analysis tool developed in our previous research [3, 4]. Our tools use Eclipse JDT and .NET Compiler Platform ("Roslyn") to parse Java and C# source code, respectively. The detection of exception flow and handling anti-patterns is implemented without using heuristics [3, 4]. 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#). We consider **both** *runtime* and *non-runtime* (i.e., both *checked* **and** *unchecked* exceptions in Java) exception flows.

3.4 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 [22]. Similarly to previous studies [36, 51], 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 3.3.2 and 3.3.3 details the traditional metrics that are used in the base model. Afterward, we construct a

[1] The exception flow analyzer and the full list of metrics with their aggregation rules, the raw data and the model construction and analysis steps scripts and results are available online at https://guipadua.github.io/eh-model-defects2018

Table 2: Exception handling flow characteristics metrics. 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.	It is more challenging for developers to handle all exceptions [3]. Missing handling exceptions is a cause for exception handlings defects. [18].
Flow Quantity - Propagated	The distinct number of possible exceptions that is propagated by the handler.	If propagated exceptions remain uncaught, there could be a risk of system failures [5, 18, 53].
Flow Quantity - Propagated and Potentially Recoverable	The distinct number of potentially recoverable possible exceptions that is propagated by the handler.	Leaving recoverable exceptions [1, 21] unhandled might increase the probability of defects since developers and users do not expect they will happen.
Flow Type Prevalence	The average prevalence of the flow exception types of a try block.	Many exception types appear in only one try block [3] and 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 [3], it might be also declared in different methods, which increases the complexity.
Flow Sources - Invoked	The average number of invoked method(s) per possible exception of a try block.	Having multiple sources for an exception creates ambiguity for developers/testers handling/testing the different possible control flow paths of such exception [3].
†Flow Sources - Documentation	The percentage of the possible exceptions of a try block found by a given exception documentation source.	Lacking immediate documentation is one of the challenges of exception handling [7, 30, 50].
†Flow Handling Strategy	The percentage of the possible exceptions of a try block that are handled with a given strategy (i.e, specific and subsumption).	The subsumption handling strategy introduces harmful uncertainty [4, 50, 53] and, therefore, could increase the probability 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 [3]).	Proper recovery actions taken during handling would reduce the probability of defects, meanwhile inappropriate actions could reveal a higher probability of defects.
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).
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.
Try Call Depth	The average relative (i.e. to the handler) call graph depth in which a possible exception was found.	The large distance between throw and catch makes the exception handling less meaningful and testing and debugging more difficult [46, 53].
†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. [7, 13, 18].

Table 3: 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 [4]).	Exception handling anti-patterns are prevalent [4] and they can compromise the robustness of the program and can lead to defects [13, 32, 35].
†Catch Recoverability	The recoverability of the exception type declared in the catch block.	Potentially unrecoverable exceptions are more challenging to handle [1, 21] and 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 and they may be related to more exceptional scenarios of the execution.
Catch Size - LOC	The number of lines of code in the catch blocks of the file.	Longer catch blocks include more code that takes measures in the event of an exception and 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).

combined model called *BSFC* by adding software metrics that are associated with quantified exception flow characteristics from prior studies [3] into the base model. We also add software metrics that are associated with the exception handling anti-patterns from prior studies [4] 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 1.

3.4.1 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

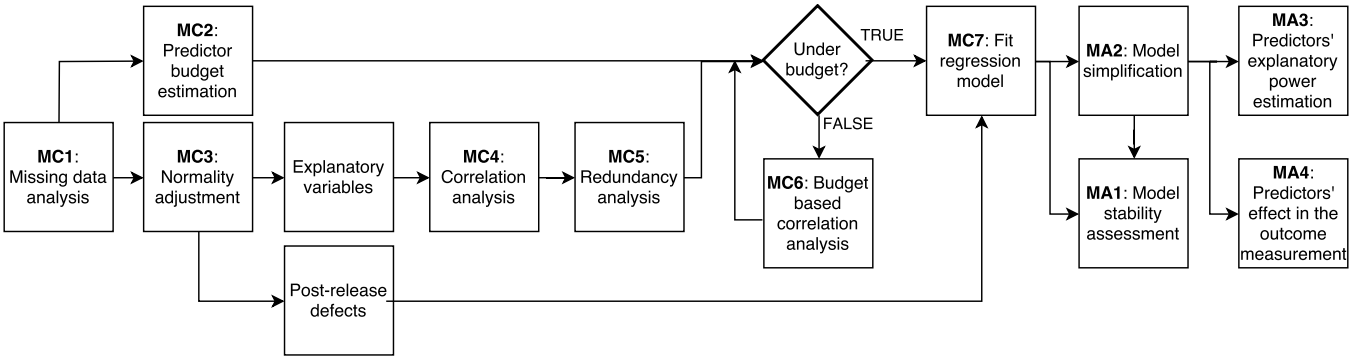

Figure 1: An overview of our modeling approach: model construction and model analysis.

cases where the file is not compilable 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 [22], we discard the files with missing data since it only stands for less than 3% of the entire data.

3.4.2 MC2: Predictors 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 [22]. To reduce the potential overfitting, one can use as a reference the amount of, at least, 15 observations per predictor, which is suggested by prior research on statistical modelling [22]. Therefore, in our study, each model will have a *# Predictors Budget* of the number of files divided by 15. The result for each project can be found in Table 4.

3.4.3 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 [36, 52]. 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.

3.4.4 MC4: Correlation analysis. Software metrics can be highly correlated to each other [19]. Highly correlated metrics (i.e., $|\rho| > 0.7$) can be clustered and then represented in regression modeling by a single predictor [22]. Prior to modeling, we evaluate the correlations among our extracted metrics. We use Spearman pair-wise rank correlation to better account for potential lack of normality in the data. We use the *findCorrelation* method from the *Caret* R Package [34]. Such method automatically removes the metrics among the highly correlated metrics with the highest mean correlation values.

3.4.5 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 [22]. This step is executed in an iterative manner in which predictors are dropped until no predictors can be predicted with an R^2 or adjusted R^2 higher than 0.9. We use the *redun* method from the *Hmisc* R Package [27]. After we perform correlation analysis and redundancy analysis, we have a list of *# Potential Predictors* for modeling, which can be found in Table 4.

3.4.6 MC6: Budget based correlation analysis. From the result of step MC2, we compare the predictor budget of each project (i.e., *# Predictors Budget* in Table 4) with, from the result of step MC5, how many potential predictors exist in the metric set (i.e., *# Potential Predictors* in Table 4). From the comparison, a given project is over budget if the number of potential predictors is higher than the budget. **If** *# Potential Predictors* is higher than the *# Predictors Budget* we execute a new correlation analysis. However, at this time we use the *# Predictors Budget* as a target in terms of the number of predictors. For example, if the *# Predictors 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 [34]. 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 [27].

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 [22]. Nevertheless, we aim to still keep the predictors that are different from each other, which could potentially contribute more to the model.

3.4.7 MC7: Fit regression model. Similar to previous work mentioned in Section 2, we use logistic regressions to model the probability 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 [28]. 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.

3.5 Model analysis

3.5.1 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 [41], and is an adequate measure for evaluating competing logistic regression models [24]. The regular

R^2 does not apply to logistic regression and deviance explained is inappropriate [22].

However, since the model is built using historical data, there is a possibility 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 [28]. 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).

3.5.2 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 [28]. 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.

3.5.3 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 [22].

3.5.4 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 probability of post-release defects. In this step, similarly to previous research [51, 52], 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 [28].

3.6 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 handling metrics we use is the number of exception handling constructs, such as try, catch or throws blocks. If such a simple metric is not significant in the models, further analysis on exception handling practices might be less promising. 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 3.3 are only meaningful if there exist exception handling constructs in the file.

4 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 a better explanation of the probability of post-release defects?

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

Table 4 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 a 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 probability of post-release defects.

The significant metric with highest χ^2 in BSFC model of Hibernate is the prevalence of particular exception types. Table 5 shows the large explanatory power of the metric on the probability 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 probability of post-release defects, while files with rare exception types have a higher probability 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 probability of post-release defects.

In Hibernate, the files with more possible exceptions handled with *Throw Wrap* action (i.e., HB-6) have lower probability 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 [3] and we find that this action is present in 55% of the catch blocks in Hibernate. On one hand, such wrapping may be used to help better explain the exception and provide more customized exception types to handle.

Table 4: A summary of the fitted models' construction and analysis.

Indicator	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

Table 5: 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 ↗) means that higher values of the metric, higher probability of post-release defects.

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

On the other hand, this pattern is often used in Java to transform a checked exception into an unchecked exception. This removes the need to handle or propagate the exception, keeping the method signatures clean. 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 probability 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 [7, 15, 50]. 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 an 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 [51].

The files with a higher percentage of handlers using the *Method* action in Hadoop have a higher probability of post-release defects

(i.e., positive relationship). The *Method* action is when other methods are called in the catch block [7, 50]. 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 probability 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 probability 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 probability 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 [3]. 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 probability 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 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 that 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 probability of post-release defects. This metric measures the number of try blocks inside another block that is not a declaration, a condition, a loop or an exception handling feature. 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 probability of post-release defects may due to the complexity of the *SynchronizedStatement*.

RQ2: Do exception handling anti-patterns contribute to a better explanation of the probability 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 probability 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 probability of post-release defects. This means that if the file has larger exception handling blocks on average, there is a higher probability 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 probability of post-release defects.

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

The percentage of catch blocks affected by the *Dummy Handler* anti-pattern has a positive relationship with the probability 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 [4, 13]. 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 the *Generic Catch* anti-pattern has a positive relationship with the probability 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 5). Prior study finds that this anti-pattern is prevalent in practice [4]. 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 probability 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 [35]. 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 probability of post-release defects.

The total number of catch blocks affected by *Log and Throw* has a positive relationship with the probability of post-release defects in Hadoop. The *Log and Throw* anti-pattern has been advocated to be harmful [35]. 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 [4] and it was found to have a small effect on the probability of post-release defects (see Table 5), practitioners should still avoid such a suboptimal practice.

5 THREATS TO VALIDITY

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

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.

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 paper. 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. Moreover, we aim to *understand* the relationship between exception handling practices and post-release defects, and we do not aim to *predict* post-release defects.

There are rooms 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.

Construct validity Our study may not cover all possible handling actions. We selected actions based on the previous research in the subject [7, 15, 50, 56, 60]. 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 that obtains both runtime and non-runtime exceptions. However, 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. Similarly, there may be other software metrics to measure other factors which could affect the results, such as, the developer's experience. Adding more metrics may provide a further understanding of its relationship with post-release defects. In addition, this paper only focuses on post-release defects as one aspect of software quality. There exist other aspects of software quality other than post-release defects. 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 [22]. 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.

6 CONCLUSION

Exception handling is an important feature in modern programming languages. Prior studies have unveiled the suboptimal usage of exception handling features in practice and have proposed exception handling anti-patterns. In this paper, 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 exception flow characteristics in Java projects have a significant explanatory power when complementing traditional software metrics in modeling post-release defect. 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.

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

REFERENCES

[1] 2017. Handling and Throwing Exceptions - .NET Framework Documentation. (mar 2017). Retrieved April 29, 2017 from https://msdn.microsoft.com/en-us/

library/5b2yeyab(v=vs.110).aspx
[2] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. 2016. How developers use exception handling in Java?. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. ACM Press, New York, New York, USA, 516–519. https://doi.org/10.1145/2901739.2903500
[3] Guilherme B. de Pádua and Weiyi Shang. 2017. Revisiting Exception Handling Practices with Exception Flow Analysis. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 11–20. https://doi.org/10.1109/SCAM.2017.16
[4] Guilherme B. de Pádua and Weiyi Shang. 2017. Studying the Prevalence of Exception Handling Anti-patterns. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 328–331. https://doi.org/10.1109/ICPC.2017.1
[5] Eiji Adachi Barbosa, Alessandro Garcia, and Simone Diniz Junqueira Barbosa. 2014. Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *2014 Brazilian Symposium on Software Engineering*. IEEE, 11–20. https://doi.org/10.1109/SBES.2014.19
[6] Rodrigo Bonifacio, Fausto Carvalho, Guilherme N. Ramos, Uira Kulesza, and Roberta Coelho. 2015. The use of C++ exception handling constructs: A comprehensive study. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 21–30. https://doi.org/10.1109/SCAM.2015.7335398
[7] Bruno Cabral and Paulo Marques. 2007. Exception Handling: A Field Study in Java and .NET. In *ECOOP 2007 âĂŞ Object-Oriented Programming*, Vol. 4609. Berlin, Heidelberg, 151–175. https://doi.org/10.1007/978-3-540-73589-2
[8] Bruno Cabral and Paulo Marques. 2011. A transactional model for automatic exception handling. *Computer Languages, Systems and Structures* 37, 1 (2011), 43–61. https://doi.org/10.1016/j.cl.2010.09.002
[9] Bruno Cabral, P. Sacramento, and Paulo Marques. 2007. Hidden truth behind .NET's exception handling today. *Software, IET* 2, 1 (2007), 233–250. https://doi.org/10.1049/iet-sen:20070017
[10] Nelio Cacho, Eiji Adachi Barbosa, Juliana Araujo, Frederico Pranto, Alessandro Garcia, Thiago Cesar, Eliezio Soares, Arthur Cassio, Thomas Filipe, and Israel Garcia. 2014. How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 31–40. https://doi.org/10.1109/ICSME.2014.25
[11] Nelio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel Garcia, Eiji Adachi Barbosa, and Alessandro Garcia. 2014. Trading robustness for maintainability: an empirical study of evolving c# programs. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* iii (2014), 584–595. https://doi.org/10.1145/2568225.2568308
[12] Tom Cargill. 1996. *C++ Gems*. SIGS Publications, Inc., New York, NY, USA. 423–431 pages.
[13] Chien-Tsun Chen, Yu Chin Cheng, Chin-Yun Hsieh, and I-Lang Wu. 2009. Exception handling refactorings: Directed by goals and driven by bug fixing. *Journal of Systems and Software* 82, 2 (feb 2009), 333–345. https://doi.org/10.1016/j.jss.2008.06.035
[14] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. 2017. Exception Handling Bug Hazards in Android. *Empirical Softw. Engg.* 22, 3 (June 2017), 1264–1304. https://doi.org/10.1007/s10664-016-9443-7
[15] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. 2008. 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*. Vol. 5142 LNCS. 207–234. https://doi.org/10.1007/978-3-540-70592-5
[16] F Cristian. 1982. Exception Handling and Software Fault Tolerance. *Computers, IEEE Transactions on* C-31, 6 (1982), 531–540. https://doi.org/10.1109/TC.1982.1676035
[17] M. D'Ambros, M. Lanza, and R. Robbes. 2010. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 31–41. https://doi.org/10.1109/MSR.2010.5463279
[18] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (aug 2015), 82–101. https://doi.org/10.1016/j.jss.2015.04.066
[19] Kalhed El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. 2001. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Trans. Softw. Eng.* 27, 7 (July 2001), 630–650. https://doi.org/10.1109/32.935855
[20] Israel Garcia and Nélio Cacho. 2011. eFlowMining: An Exception-Flow Analysis Tool for .NET Applications. In *2011 Fifth Latin-American Symposium on Dependable Computing Workshops*. IEEE, 1–8. https://doi.org/10.1109/LADCW.2011.18
[21] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. ChapterÂă11.ÂăExceptions - Java SE Specification. (feb 2015). Retrieved March 29, 2017 from http://docs.oracle.com/javase/specs/jls/se8/html/jls-11.html
[22] Frank E. Harrell ,. 2015. *Regression Modeling Strategies*. Springer Series in Statistics, Vol. 64. Springer International Publishing, Cham. 501–507 pages. https://doi.org/10.1007/978-3-319-19425-7
[23] Ahmed E. Hassan. 2009. Predicting Faults Using the Complexity of Code Changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 78–88. https://doi.org/10.1109/ICSE.2009.5070510
[24] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.
[25] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. 2013. Mining the relationship between anti-patterns dependencies and fault-proneness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 351–360.
[26] Jang Wu Jo, Byeong Mo Chang, Kwangkeun Yi, and Kwang Moo Choe. 2004. An uncaught exception analysis for Java. *Journal of Systems and Software* 72, 1 (2004), 59–69. https://doi.org/10.1016/S0164-1212(03)00057-8
[27] Frank E Harrell Jr. 2017. Hmisc: Harrell Miscellaneous. (2017). https://CRAN.R-project.org/package=Hmisc
[28] Frank E Harrell Jr. 2017. rms: Regression Modeling Strategies. (2017). https://CRAN.R-project.org/package=rms
[29] Maria Kechagia, Tushar Sharma, and Diomidis Spinellis. 2017. Towards a Context Dependent Java Exceptions Hierarchy. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 347–349. https://doi.org/10.1109/ICSE-C.2017.134
[30] Maria Kechagia and Diomidis Spinellis. 2014. Undocumented and unchecked: exceptions that spell trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, New York, New York, USA, 312–315. https://doi.org/10.1145/2597073.2597089
[31] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. 2016. Examining programmer practices for locally handling exceptions. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. ACM Press, New York, New York, USA, 484–487. https://doi.org/10.1145/2901739.2903497
[32] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. *Empirical Softw. Engg.* 17, 3 (June 2012), 243–275. https://doi.org/10.1007/s10664-011-9171-y
[33] Sunghun Kim, Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, and Shivkumar Shivaji. 2013. Predicting Method Crashes with Bytecode Operations. In *Proceedings of the 6th India Software Engineering Conference (ISEC '13)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/2442754.2442756
[34] Max Kuhn. 2017. caret: Classification and Regression Training. (2017). Retrieved November 29, 2017 from https://CRAN.R-project.org/package=caret
[35] Tim McCune. 2006. Exception handling antipatterns. (apr 2006). Retrieved February 28, 2017 from https://community.oracle.com/docs/DOC-983543
[36] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Softw. Engg.* 21, 5 (Oct. 2016), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9
[37] P. M. Melliar-Smith and B. Randell. 1985. Software Reliability: The Role of Programmed Exception Handling. *Reliable Computer Systems* (1985), 143–153. https://doi.org/10.1007/978-3-642-82470-8
[38] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. 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)*. ACM, New York, NY, USA, 181–190. https://doi.org/10.1145/1368088.1368114
[39] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* (2017), 1–35. https://doi.org/10.1007/s10664-017-9512-6
[40] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 452–461. https://doi.org/10.1145/1134285.1134349
[41] Nico JD Nagelkerke et al. 1991. A note on a general definition of the coefficient of determination. *Biometrika* 78, 3 (1991), 691–692.
[42] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. 2016. Analysis of exception handling patterns in Java projects. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. ACM Press, New York, New York, USA, 500–503. https://doi.org/10.1145/2901739.2903499
[43] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. 2018. 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 (2018), 1 – 18. https://doi.org/10.1016/j.jss.2017.10.032
[44] Haidar Osman, Andrei Chiş, Claudio Corrodi, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Exception Evolution in Long-lived Java Systems. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 302–311. https://doi.org/10.1109/MSR.2017.21

[45] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, Experience and Defects: A Fine-grained Study of Authorship. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 491–500. https://doi.org/10.1145/1985793.1985860

[46] Darell Reimer and Harini Srinivasan. 2003. Analyzing exception usage in large java applications. In *Proceedings of ECOOPâĂŹ2003 Workshop on Exception Handling in Object-Oriented Systems*. 10–18.

[47] Martin P Robillard and Gail C Murphy. 1999. 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*. 322–337. https://doi.org/10.1145/318773.319251 arXiv:arXiv:1011.1669v3

[48] Scitools.com. [n. d.]. Understand: Visualize Your Code. ([n. d.]). Retrieved November 29, 2017 from https://scitools.com/

[49] Scitools.com. [n. d.]. What Metrics does Understand have? ([n. d.]). Retrieved November 29, 2017 from https://scitools.com/support/metrics_list

[50] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the exception handling strategies of Java libraries. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. 212–222. https://doi.org/10.1145/2901739.2901757

[51] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering* 20, 1 (feb 2015), 1–27. https://doi.org/10.1007/s10664-013-9274-8

[52] Emad Shihab, Christian Bird, and Thomas Zimmermann. 2012. The Effect of Branching Strategies on Software Quality. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '12)*. ACM, New York, NY, USA, 301–310. https://doi.org/10.1145/2372251.2372305

[53] S. Sinha, A. Orso, and M.J. Harrold. 2004. Automated support for development, maintenance, and testing in the presence of implicit flow control. In *Proceedings. 26th International Conference on Software Engineering*. 336–345. https://doi.org/10.1109/ICSE.2004.1317456

[54] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan. 2013. Predicting Bugs Using Antipatterns. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, Washington, DC, USA, 270–279. https://doi.org/10.1109/ICSM.2013.38

[55] Suresh Thummalapenta and Tao Xie. 2009. Mining Exception-handling Rules As Sequence Association Rules. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 496–506. https://doi.org/10.1109/ICSE.2009.5070548

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

[57] Benwen Zhang and James Clause. 2014. Lightweight automated detection of unsafe information leakage via exceptions. *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014* (2014), 327–338. https://doi.org/10.1145/2610384.2610412

[58] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2014. An Empirical Study of the Effect of File Editing Patterns on Software Quality. *J. Softw. Evol. Process* 26, 11 (Nov. 2014), 996–1029. https://doi.org/10.1002/smr.1659

[59] Lingli Zhang and Chandra Krintz. 2009. As-if-serial exception handling semantics for Java futures. *Sci. of Computer Programming* 74 (2009), 314–332. https://doi.org/10.1016/j.scico.2009.01.006

[60] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 415–425. https://doi.org/10.1109/ICSE.2015.60