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

Srikrithi Chamarthi, Anusha Yeramalla, Bala Sharanya Devarapu
Dept. of Engineering and Computer Science, Concordia University, Montréal, Québec, Canada

***Abstract*—This paper is a replication of a previous work, 'Studying the Relationship between Exception Practices and Post-release Defects'. Exceptions are one of the common events that are often seen in programming. Though the code to handle these exceptions is kept separately, often this code leads to many issues. Hence, in this paper the relationship between the quality of software in terms of post-release bugs and the exception handling metrics has been observed. The observations were performed on the open source software Apache KAFKA. After deriving the required pre-release defects, post-release defects, exception flow characteristics and exception anti-pattern metrics, models have been constructed. Depending on the results from the models, it was found that exception-flow characteristics have a remarkable relationship with post-release defects. Hence, it can be understood that the IT firms need to allocate more teams to perform those tasks to avoid any catastrophic failures in future.**

I. A brief overview on "Studying the Relationship between Exception Practices and Post-release Defects"

Many Catastrophic software failures are due to incomplete knowledge of the developers while handling exceptions. Also, there is not much work how this misuse of exceptions is going to affect the software quality, which is one of the reasons why developers are not taking an initiative to avoid anti-patterns. Hence, this paper which is the first to provide the relationship between software quality and exception practices suggests to allocate more resources to improvise exception handling besides highlighting the need for techniques in handling anti-patterns. In this paper, the authors have worked on two java projects and one C# project because of their popularity and their presence in the previous studies. Using these three projects, authors tried to answer two research questions, namely ", "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?" [9]. After extracting the four types of metrics (post-release defects, Traditional product metrics, traditional process metrics and exception handling), models were constructed to extract the results. A BASE model in which only traditional metrics were used, BSFC which is combination of BASE model and flow characteristics, and BSAP, a combination of BASE model and anti-patterns are all the models that were used in the paper. Using the above datasets and following seven model construction steps, a model has been built and analyzed. Few of the paper's important findings are that for Java projects, exception flow characteristics along with traditional metrics can model post-release defects. Dummy Handler, Generic Catch anti-pattern, Ignoring Interrupted Exception, Log and Throw are all the anti-patterns that are found to have positive relationship with respect to post-release defects. Hence, this study suggests to avoid at least those anti-patterns

II. Subject Projects

Apache KAFKA has been considered for this study. KAFKA has been developed using both Java and Scala. But, for this study only Java files were taken into consideration. Though the latest version of Apache KAFKA is 3.4.0, we have considered 3.2.0 because of its stability and also to make sure we have enough pre-release defects and post-release defects to work on. There is also an advantage of the availability of the Issue details that can be extracted using an issue tracking system called Jira. Compared to the latest version, v3.2.0 has more number of pre release bugs and post release bugs which were used for the current study.

TABLE I
AN OVERVIEW OF THE SUBJECT PROJECTS

Project Characteristics	
Name of the Project	KAFKA
Language	Java
Purpose	Real-time data processing
Release Version (tag name)	3.2.0v
Release Version Date	May 17, 2022
Latest Post Release Version (tag name)	3.4.0
Latest Release Version Date	Feb 7, 20223
# Files	3,421
# SLOC (K)	489,062
#Classes	5715
# Pre Release Defects	184
# Post Release Defects	75
# Files with Post Release Defects	4137

III. METRICS

Before building the models to analyze, we extract four categories of metrics based on the considered Java KAFKA project. The four categories are:

A. *Post-release defects:*

To obtain the number of post-release defects for each file, we utilized Jira to extract the relevant data based on the release date and version. All the resolved and closed defects that occurred after the version release were considered as post-release defects. Subsequently, we used Git Command Tool to extract a log file from the Kafka folder using git log. These two files were combined to produce a csv file. The process of merging the files was carried out using a specific code, which is available in our Git repository for reference.

B. *Traditional Product Metrics:*

Product metrics, such as size and Cyclomatic complexity, have been found to be the most effective measure for post-release defects. In order to obtain these metrics, a static code analysis tool called Understand was utilized to calculate various product metrics, including but not limited to Average Cyclomatic, Average Line, Average Line Blank, Average Line Comment, Count Line, and Count Line Blank.

C. *Traditional Process Metrics:*

Process metrics play a crucial role in measuring the effectiveness of a software development process. In our study, we have focused on two process metrics, namely change metrics and pre-release defects.

1) Change metrics: : Change metrics are calculated using the pre-release changes i.e., all the changes that have been made prior to the release are taken into consideration. Since, the current 3.2.0v is released on May 17, 2022, we have considered all files from May 17, 2022 to September 21, 2021. We have used git bash to loop through each java file to extract total number of changes in the code and code churn. We calculate these metrics in our Kafka 3.2.0 branch Git repository by writing a script extracting the dates of the release of our selected version to a previous version of our choice and extracting the two metrics. The output is saved in a csv file with file path, total changes, and code churn for each java file. The script uses Git commands and awk to extract the necessary information. With these we get two metrics i.e., the changed lines of code and the code churn, which are saved under Process metrics.

2) Pre-release quality metrics : : Similar to post-release defects, for pre-release defects we have used Jira to extract all the issue ids that were raised prior to release of the version and have been resolved and closed. After extracting the issue ids we have combined this with the git log files to extract the file name and the number of pre-release bugs.

D. *Exception Handling Metrics*

There are two types of exception handling metrics, namely Exception Flow Characteristics metrics and Exception handling anti-pattern metrics

1) Exception flow characteristics metrics: : Flow metrics are one of the important metrics that are considered to find how software quality is being affected. These metrics are used to identify sub-optimal exception handling practices, which can lead to poor system performance and stability. This metrics describes the characteristics of exception flow[reference]. TABLE II describes the flow metrics that have been considered for this paper. These metrics are obtained from the KAFKA repository using ASTParser that been uploaded in our project Git repository. The Exception flow metrics that are considered for the project are given discussed below in the table.

TABLE II
FLOW METRICS

Flow Metric	**Description**
Try Quantity	Total number of try blocks in each file were found for this metric, by iterating over all the try blocks in the file.
Try LOC	Lines Of Code is found for each try statement and summed up to find the total LOC for each file.
Try SLOC	Source Lines Of Code is found for each try statement and summed up to find the total SLOC for each file.
Invoked Methods	The total number of method calls in every Try block is calculated
Flow Handling Strategy	*Specific Handling Strategy:* The exception has its specific catch block to handle the exception. *Subsumption Handling Strategy:* The exception is handled by its superclass. Code has been written to identify each case by comparing the thrown exception and the caught exception.
Flow Handling Actions	The total number of possible exceptions and the percentage of possible exceptions of a try block handled with ”nested try”,”return”,”continue” actions[2] is being calculated and the metrics are collected by detecting the ”nested try”,”return”,and ”continue” instances in every file.
Catch Quantity	Number of catch blocks are counted for each file.
Catch Size - LOC	Total LOC for all the catch blocks is generated.
Catch Size - SLOC	Total SLOC for all the catch blocks is generated.
Catch Recoverability	We have recoverable and unrecoverable exceptions. If an exception has a catch block to handle it, then it is referred to as a recoverable exception, else unrecoverable.
Catch Anti-patterns	The total number of handlers affected and the percentage of handlers affected for every anti-pattern that is considered as described in the TABLE III

2) Except handling anti-pattern metrics: : Anti-patterns are known to be one of causes for program failures and hence they need to be avoided. In this paper, we have concentrated on 4 types of anti-patterns namely, Throws Kitchen Sink, Log and Throw, Nested Try and Destructive wrapping.

AntiPatterns

A. *Throws Kitchen Sink* anti-pattern involves throwing multiple exceptions from a method without proper differentiation between them. This anti-pattern can be avoided by wrapping multiple exceptions that have similar meanings into a single checked exception. To detect this anti-pattern in Java projects, an ASTVisitor class from the *org.eclipse.jdt.core* package is utilized. A visitor class is created to find method declarations and check if there are multiple thrown exceptions. A set of exceptions is used to keep track of which exceptions were used, and conditional statements are recursively checked to determine if the exception was thrown inside them. If any of the exceptions were not used, the case is identified as a ”throws kitchen sink” anti-pattern.

B. *Log and Throw* anti-pattern occurs when an exception is both logged and thrown in a program. This can lead to confusion for someone trying to understand the logs, and it is recommended to either log or throw an exception but not both. To detect this anti-pattern, a LogandThrowVisitor class is used which extends the ASTVisitor Class. The visit (MethodInvocation) and visit(CatchClause) methods of the ASTVisitor are overridden. If both the log statement and the throw statement are found in the catch clause, then the ”Log and Throw” anti-pattern is identified.

C. *Destructive Wrapping* anti-pattern occurs when a higher-level exception is thrown that wraps a lower-level exception without providing any additional information. This can make it difficult to identify the root cause of the problem and lead to confusion and inefficiencies. The Visitor class can detect this anti-pattern by traversing the AST of the Java code and searching for catch blocks that catch exceptions of a specific type. The class looks for ThrowStatement nodes within these catch blocks that represent throw statements used to throw exceptions. If a ThrowStatement node is found within a catch block, the class checks the type of the thrown exception. If the type of the thrown exception does not match the type of the caught exception, it is assumed that destructive wrapping is identified.

D. *Nested Try* anti-pattern is characterized by nested try statements in Java code. If a TryStatement node is found, the Visitor class retrieves the parent node and checks if it is also a TryStatement it is detected as Nested Try antipattern.

IV. Model Construction

Following the extraction of necessary CSV files, three types of models - BASE, BSFC, and BSAP - were built by combining the files appropriately. The BASE file includes all product metrics and process metrics from the Understand tool, while the BSFC file is created by merging BASE with Flow metrics data. Finally, the BSAP file is produced by merging BASE with Anti-pattern data.

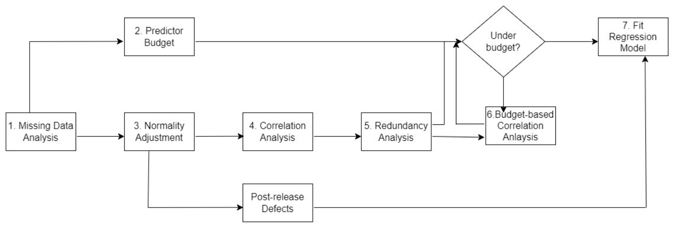

Fig. 2. Overview of Model Construction.

Step 1: Missing Data Analysis

The merged files were later pre-processed by filtering out files other than the one's written in Java and removing null or missing data. The data was also filtered according to the research requirements.

Step 2: Predictors budget estimation:

To avoid an overfitted model, we have calculated Predictors Budget by deciding the total number of files by 15. We had 4136 files in total, which gave us a #Predictors budget of 275 after dividing the total files by 15

Step 3: Normality adjustment:

In some cases, the data was found to be non-normally distributed. To address this, we applied a log transformation ($\log_{10}(x+1)$) to our metrics to reduce skewness and achieve normal distribution. We determined which metrics required transformation by calculating their skewness and applying transformation to those with skewness greater than 1. However, during model building, we found that metrics with or without transformation gave similar results.

Step 4: Correlation Analysis:

Firstly, we identify and remove variables with zero variance from the dataset. Then we calculate the correlation matrix of the remaining variables using the Spearman method. Variables with high correlation (greater than 0.75) are removed to reduce redundancy. Finally, a new dataset with the remaining variables is created for further analysis. This pre-processing step is important to ensure that the data used in the analysis is of high quality and not redundant.

Step 5: Redundancy Analysis :

After finding the columns with low correlation, redundancy analysis is further performed to avoid any redundancy. Sometimes,one predictor can be explained in terms of other, hence redundancy analysis is performed to remove all those metrics and an adjusted R^2 higher than 0.9. Now, after eliminating the redundant columns we then have #Potential Predictors

Step 6: Budget Based correlation analysis:

We have compared the #Predictor budget with the #Potential Predictors budget and made sure that we didn't cross the budget.

Step 7: Fit Regression Model:

After identifying the non-correlated and non-redundant columns, a logistic regression model was developed to determine the likelihood of having post-release defects in a file, while ensuring that the model remains within the budget. Zero variance was checked before building the model to prevent over fitting. The glm() model was used instead of lrm() to fit the Kafka data set, with the family set as 'binomial' since the response variable, 'Post.release.Bugs', is not binary. The data was split into training and testing sets using an 80:20 data partition.

TABLE III
A summary of the fitted models' construction and analysis.

Indicator		Kafka	
	BASE	BSFC	BSAP
# Predictors Budget	15	15	15
# Potential Predictors	18	25	20
Adjusted Correlation Cutoff	0.75	0.75	0.75
Optimism-reduced Nagelkerke R^2	0.0896	0.0934	0.0904

V. Model Analysis

This section outlines the process of analyzing a logistic regression model used to predict the likelihood of post-release defects in software, based on historical data. The analysis involves four steps:

Model stability assessment: To evaluate the accuracy of the model, the Nagelkerke-R2 statistic is used, which is adjusted to range from 0 to 1. Due

TABLE IV
SIGNIFICANT METRICS IN THE FINAL MODELS WITH WALD χ^2 AND EFFECT VALUES

ID	Metrics		Base	BSFC		BSAP	
			χ^2	χ^2	Effect	χ^2	Effect
KA-1	Changes	↗	5.53	4.24	4.8%	6.2	5.0%
KA-2	Catch Anti-pattern (Destructive wrapping)	↘				5.5	1.5%
KA-2	Catch Anti-pattern (Nested Try)	↘				4.8	0.8%
KA-3	Try Scope	↗					
KA-4	Flow Handling Actions (Nested try)	↘		4.56	-3.8%		
KA-5	Catch Recoverability, Catch Quantity, Catch Size (LOC and SLOC)	↗				5.5	1.5%
KA-6	Flow Sources (Invoked)	↗		8.46	3.8%		

to software limitations, 100 repetitions of Bootstrap are used instead of the original paper's 1,000 to ensure model stability and validate its performance. An optimism-reduced Nagelkerke-R2 is calculated to account for predictor noise and different data samples. Nagelkerke-R2 was calculated by subtracting the null-variance with the variance of the model and later dividing it with the null-variance. This Optimized nagelkerke-R2 is further calculated by subtracting the nagelkerke-R2 with the mean of the variation of the data set derived from the bootstrap method to estimate the model's R-squared values by repeatedly re-sampling the data with replacement and fitting a multinomial logistic regression model to each sample.

*Model simplification:*Insignificant predictors are iteratively removed from the model using the fast backward predictor selection technique until only significant predictors remain. A significance level of 0.05 is used as the stopping rule. In our project, since glm() is used to build a model, model simplification is performed using backward selection with 'stepAIC'. The resulting simplified model is printed with a summary of its coefficients. This simplified model is further used in the next step as an input.

Predictors' explanatory power estimation: The Wald 2 test is used to identify the predictors with the highest explanatory power among the significant predictors. This is done by performing backward selection to simplify the logistic regression model and performing a Wald Chi-Square test on the significant predictors to determine their significance. If significant predictors are found, their Chi-Square values and p-values are printed in a table sorted in descending order of Chi-Square values. If no significant predictors are found, a message is printed indicating the same. A higher Wald 2 indicates a higher contribution to the model fit. This is further used to assess our model to be a better fit.

Predictors' effect in the outcome measurement: The impact of each significant predictor on the model outcome is measured by setting all predictors at their mean value and increasing the value of each predictor by 10% while keeping all other significant predictors at their mean values. The differences in the model outcome are measured as the effect of the predictor. This is used to find the effect of a single factor on our model.

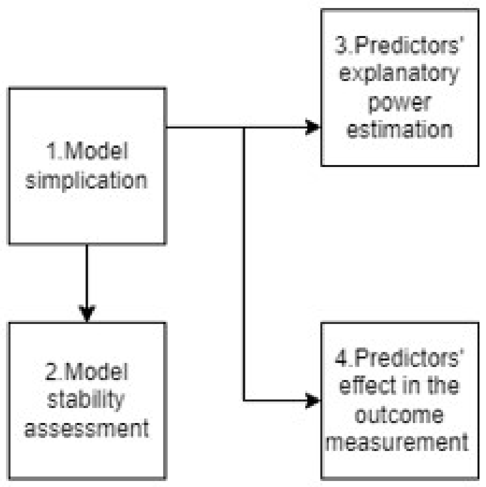
1.Model simplication
2.Model stability assessment
3.Predictors' explanatory power estimation
4.Predictors' effect in the outcome measurement

Fig. 5. Overview of Model Analysis

VI. CASE STUDY RESULTS AND DISCUSSION

In this section we put forth the results observed by analyzing the model that was constructed and also draw a comparison between the model built and the original paper [1] that was used to replicate

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.

After looking at the results from table *III*, exception flow metrics were found to show more impact on the BSFC model. These flow metrics made the model a better-fit. The Nagelkerke R^2 calculated for the BSFC was higher than that of the BASE model, which acts as an indicator that the flow metrics make the model a better fit, by explaining the post-defects. Also, our findings are similar to that of the original paper where the R^2 value was higher for flow metrics.

The flow handling actions may have a statistically significant relationship with the probability of post-release defects.

In the analysis, it was observed that the frequency of flow handling actions has a negative correlation with the occurrence of post-release defects. This means that a higher frequency of such actions is associated with lower post-release defects. Additionally, the presence of exception handlers was found to reduce the post-release defects.

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

From our study we found that the files with the existence of invoked methods also have a positive relationship with post-release bugs. It is really important to know from where an exception is generated. Without having enough information about the origin of the exception, it is difficult to handle it properly, which eventually results in more post-release defects. From table *IV* KA-7 shows that existence of invoked methods that result in post-release defects.

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.

The Exception handling anti-patterns can provide useful information about the likelihood of post-release defects, they are not necessarily reliable predictors of those defects. Our study found that in most cases, the anti-patterns did not have a statistically significant impact on post-release defects. This means that while the anti-patterns may be worth considering as part of an overall analysis of software quality, they should not be relied on as the sole indicator of post-release defects.

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

As per the results in TABLE IV,we can say that the larger catch blocks are associated with a higher likelihood of post-release defects. This may be explained by the idea that larger catch blocks may indicate more complex code that is more difficult to debug or maintain, which can increase the likelihood of errors or defects. However, Our study found that the size of catch blocks was not highly correlated with other file size metrics, indicating that it provides unique information that can help explain the likelihood of post-release defects.

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

The Exception handling anti-patterns are more likely to be associated with post-release defects than others. For example, Our study found that the Destructive Wrapping and Nested Try anti-patterns in Kafka were all associated with a higher likelihood of post-release defects. These anti-patterns indicate situations where exceptions are not handled properly or effectively, which can lead to errors or defects down the line. Developers should be aware of these anti-patterns and take steps to avoid them in their code to reduce the likelihood of post-release defects.

VII. CONCLUSION

This study aimed to investigate the relationship between exception handling practices and anti-patterns with post-release defects in software. The study replicated the findings of research paper[1] and found that exception flow characteristics in Java projects have a significant impact on post-release defects, complementing traditional software metrics. This result underscores the importance of proper exception handling. While most exception handling anti-patterns did not show a significant relationship with post-release defects, some anti-patterns had a positive relationship, and developers should avoid them. Overall, this study emphasizes the need to avoid suboptimal exception handling practices and implement techniques that improve exception handling in software development practice.

VIII. REFERENCES

[1] M. Wiang, *"Studying the Relationship between Exception Handling Practices and Post-release Defects,"* in IEEE Transactions on Software Engineering, vol. 45, no. 10, pp. 993-1014, Oct. 2019, doi: 10.1109/TSE.2018.2872144.

[2] S. Padua and W. Shang,*"Revisiting Exception Handling Practices with Exception Flow Analysis,"* in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, Jul. 2018, pp. 84-94, doi: 10.1145/3213846.3213863.