EH-Recommender: Recommending Exception Handling Strategies Based on Program Context

Yuhang Li, Shi Ying, Xiangyang Jia, Yisen Xu, Lily Zhao, Guoli Cheng, Bingming Wang, Jifeng Xuan
School of Computer Science, Wuhan University
Bayi Road 299, Wuhan, China
{lyh2012, yingshi, jxy, xys10010, lilyzhao127, chengguoli, wbingming, jxuan}@whu.edu.cn

***Abstract*—Exception handling is widely used in software development to guarantee code robustness and system reliability. Developers are expected to choose appropriate handling strategies to ensure exceptions are handled properly without causing program crashes or unintended behaviors. However, making such choices is challenging especially for the novices due to lack of experience on exceptional flow design. To assist developers in deciding how to handle exceptions, we propose a method to automatically recommend exception handling strategies based on program context. This method learns practices of exception handling from existing high-quality projects and code by well-skilled developers. We extracted three types of program context (exceptional context, architectural context, and functional context) as features and applied machine learning techniques to recommend an optimized strategy of exception handling. We conducted the evaluation on 10 open source Java projects. Experimental results show that our approach reaches high prediction accuracy in choosing exception handling strategies.**[1]

***Index Terms*—Exception Handling; Machine Learning; Recommendation**

I. Introduction

Exception handling is an effective way to enhance code robustness and program reliability [1]. Most modern programming languages (e.g., C++, C#, Java, Python) have a built-in exception handling mechanism to guide (sometimes force) developers to consider exceptional paths of the program, thus reduces the probability of program crashes, and provides necessary information (e.g., stacktraces) for troubleshooting [2].

When exceptions are raised at some program locations, developers need to decide how to handle them, i.e., to make sure exceptions are propagated, logged or re-wrapped correctly and finally handled at a right place [3]. These decisions are crucial to program quality. The study by Marinescu et al. [4] shows that classes with an improper manner of exception handling might result in a higher probability of exhibiting defects than classes with an proper manner. An empirical study of Coelho et al. [5] also shows that low-quality exception handling can induce many errors to the source code.

However, handling exceptions properly is not an easy job. Developers have to make a series of decisions during exception handling: (1) should the exception be caught at the current method or be thrown to the caller method? (2) If we catch the exception, should we handle it (e.g., recover the program state, or release the resources) right here or not? (3) If we don't handle it, should we re-throw the exception or just simply ignore it? (4) If we re-throw the exception, should we wrap the error message into a new exception, or just throw it as it is?

To make these decisions, developers need to know where is the right place to solve the exception, and where to log enough yet precise error information to assist troubleshooting. At the same time, developers should ensure that these decisions do not introduce further bugs, or hurt the understandability and readability of the program. Most time, balancing all these factors heavily depends on the developer's experience, and there are no formal regulations or programming instructions that can be easily followed [6].

Therefore, for many developers, handling exceptions is an intractable issue. Empirical studies [7], [8] show that even the most experienced developers can make mistakes in dealing with exceptions, just like the novices. Such situations are quite common that developers either misuse the exception handling features or just neglect them, which leads to inefficient programming. And moreover, improper exception handling code can inadvertently increase the risk of program error, because they are the least understood and tested code in the software system. A study by Sawadpong et al. [9] shows that the defect density of exception handling constructs is approximately three times higher than overall.

To help developers deal with exceptions more effectively, we propose an approach named EH-Recommender, which can recommend the exception handling strategies (THROW, HANDLE, LOG&IGNORE, WRAP&RETHROW) automatically by learning from existing good exception handling practices.

EH-Recommender approach firstly extracts exception handling code fragments from well-known high quality open source projects, eliminates noise data, and then automatically identifies and labels the exception handling strategy for each code fragment.

After that, the approach extracts a list of features of these code fragments for training. These features try to cover the program contexts which might influence the exception handling decisions, including the *exceptional context* which represents the cause of the error, the *architectural context* which indicates the logical layer of current method, and the *functional context* which represents the functionality of code fragments.

[1]Shi Ying and Xiangyang Jia are corresponding authors.

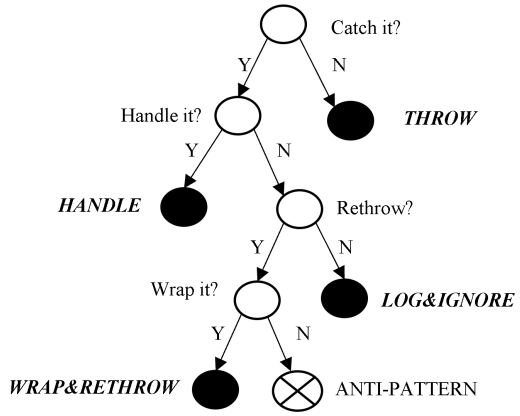

Fig. 1. Decision Process and Exception Handling Strategies

Then the machine learning techniques (e.g., SVM, Random Forest, or CNN) are used to train the labeled and featured exception handling data. As a result, a model is learned, which can be used to recommend appropriate exception handling strategies for new code fragments with diverse program contexts.

We conduct a set of experiments on 10 high quality open source Java projects, which have more than 90,000 exception-handling code fragments. The result shows that our method gains high precision in recommending the exception handling strategies. It shows a good prospect of application in improving development efficiency as well as the quality of exception handling code.

The structure of this paper is organized as follows. Section II introduces the motivation of this work. Section III describes the detailed approach of our method. Section IV reports the experiment we have done and the evaluation results. We discuss the limitations in Section V and the related works in Section VI. In Section VII we conclude this paper.

II. Motivation

A. Exception Handling Strategies

In this paper, exception handling strategies refer to the manners to handle exceptions based on specific program context. Fig. 1 illustrates the common decision-making process of exception handling, which produces four typical decisions: THROW, HANDLE, LOG&IGNORE, WRAP&RETHROW. Therefore, we divide the exception handling strategies into four types as well in this paper accordingly (as shown in Table I).

As shown in Fig. 1, when an exception is triggered by a method invocation, developers are expected to find an appropriate way to handle it. The first question to consider is, whether we should catch this exception or not? If the code does not catch the triggered exception, then we define such an exception handling strategy as THROW. This is the most common exception handling manner. According to our statistics on ten target open source projects, the ratio of directly throwing exception to catching exception is about 7:3.

If we decide to catch the exception, we should consider that how should we deal with the exception inside the catch block. Thus, the second decision that we need to make is: whether the exceptional situation should be handled (e.g., recover the program state, or release resources) in current catch block? If the answer is yes, we identify this kind of exception handling strategy as HANDLE.

If we catch the exception but do not handle it, we need to make a further decision: whether we need to re-throw the exception? If yes, we should wrap the original exception information as a new exception and re-throw it. The wrapped exception contains additional error information from the current method besides the information of the original exceptions. We identify this kind of exception handling strategy as WRAP&RETHROW. An example of WRAP&RETHROW can be like this: a database access method throws an exception named *DataAccessException*, and an order management method catches this exception, inside the catch block the exception object was wrapped into a new exception named *DublicateOrderException* and to be thrown to its caller method. Directly re-throw the original exception is a typical anti-pattern of exception handling (see Section III-B), so it is not a valid strategy in this paper.

If an exception is caused by a tolerable error, and we decide not to handle it or re-throw it, we can simply ignore it without further operations more than log the error information. We identify this kind of exception handling strategy as LOG&IGNORE. That's also reasonable, because for some types of exceptions or exceptions under specific context, ignoring them has little impact on the system running. For example, when we import a set of data, but some records have errors, we do not need to stop the import process immediately. Instead, we simply log the error records and continue the import process.

These four strategies are high-level decisions on handle exceptions. After choosing a strategy, developers need further finer-grained decision-making to handle exceptions, such as which program state should be reset? Or which information should be recorded into logs? These finer-grained decisions are not taken into consideration in this paper.

B. Why We Need Multiple Exception Handling Strategies?

Exceptions are ubiquitous in software system, and the root causes of exceptions are various. Accordingly, exceptions triggered by different reasons and different program context should be handled with different strategies. For example, exceptions caused by programming errors should be logged to facilitate debugging; exceptions caused by invalid input from a client method should be thrown to the caller method; exceptions related to resources and global states should be handled by recovering and cleaning-up.

Considering the diversity of exceptions, it is not reasonable to throw all exceptions to the caller method. If the exception, which is intended to be handled, is not handled timely and properly, the program might occur resource leaks, or inconsistent states, and leads to improper program behaviors or system crashes. On the other hand, it is not feasible to

TABLE I
STRATEGIES OF EXCEPTION HANDLING

Strategy	Description	Examples from open source projects
THROW	Throw an exception directly without catching it.	public String readFile (String path) throws IOException{ … }
HANDLE	Catch the exception and handle it, e.g., recover the program from exceptional paths, or release resources etc.	catch (InterruptedException e) { executor.shutdownNow(); Thread.currentThread().interrupt(); }
LOG&IGNORE	Catch exceptions and record the error information for troubleshooting.	catch (SocketException e) { log.warn("Error while setting soTimeout to 60000"); }
WRAP&RETHROW	Catch exceptions and wrap it as a new exception with additional information.	catch (IOException e) { throw new IllegalStateException("Unable to scan file", e); }

catch and handle every exception either. If an exception, which is intended to notify the error to the client method, is swallowed immediately, the client method may get an incorrect result rather than an error notification. These facts make troubleshooting around exception handling extremely difficult.

Thus, developers must use multiple handling strategies to cope with exceptions according to the program context.

C. Why the Exception Handling Decision Is Not Easy?

Except for the diversity of exceptions, we further explore the reasons that preventing developers from making proper exception handling decisions. We list 4 major reasons as follows:

- In order to handle exception properly, developers are expected to analyze the program context, and clearly understand the effect of different handling strategies on the maintainability, understandability, and performance of the program. This is a difficult work for novices who have little experience of design.
- A software system typically contains multiple logical layers or several sub-modules, and it is common that the overall exception flow should be carefully designed to handle exceptions [10]. Therefore, the holistic knowledge of the system structure is necessary. However, most time, the design of exception flow is lacked or not well documented. In an empirical study [11], 61% of respondents said that documenting exception handling is not taken into consideration seriously during the design phase.
- In addition, the lack of rigorous rules or context-specific instructions [6] (which is also hard to formulate such regulations) makes it confusing for developers to choose appropriate exception handling strategies. Empirical study shows [11], only 27% of respondents claimed that the policies and standards for implementing exception handling were part of their organizational culture.
- In many companies, the priority of dealing with exception handling is generally inferior to the implementation of functions. As a result, when the scale of the software system expands, these poor exception handling code will cause many strange bugs that are hard to fix [7].

In a word, exception handling heavily depends on the experience of developers, but the experience is hard or almost impossible to be documented. This is a main problem blocking developer to handle exception properly. To solve this problem, we propose an approach to learn the experience of developers from high-quality projects via training a machine learning model. This approach can help developers make better decisions of exception handling strategies.

III. APPROACH

We propose an approach, named EH-Recommender, to leverage the existing exception handling experience by learning good exception handling practices from high-quality projects.

A. The Workflow of EH-Recommender

Fig.2 illustrates the workflow of EH-Recommender approach, which has two stages: training and recommending. The training stage is to construct the classification model while the recommending stage is to exploit the model to determine which exception handling strategies are appropriate to handle exceptions raised from new code instances.

Sample collection : The first step of the workflow is to obtain all the exception handling code fragments from massive source code. In this paper, we focus on *checked exception*s, i.e., the exceptions which must be handled explicitly in the program. Because unchecked exceptions (such as errors, or RuntimeExceptions in Java language) are often caused by unrecoverable fault, and they are discouraged to be handled explicit during programming. We select out all code instances that contain exception handling with their additional contextual information, including the name and comment of their packages, classes and methods.

Labeling: In the second step, each code fragment is identified and labeled with one of the exception handling strategies (THROW, HANDLE, LOG&IGNORE, WRAP&RETHROW) by statically analyzing the code fragments and the method declaration. In order to get high-quality training data, we need to filter out the noises—the exception handling code conforming to well accepted exception handling anti-patterns.

Feature extraction: In this step, the contextual features are extracted from exception handling instances. The features represent three types of contexts that influence the exception handling decision-making: (1) exceptional context, which indicates the cause of the error, including the exception type, the parent exception classes and its comments; (2) Architectural context, which represents the logical layer of current method,

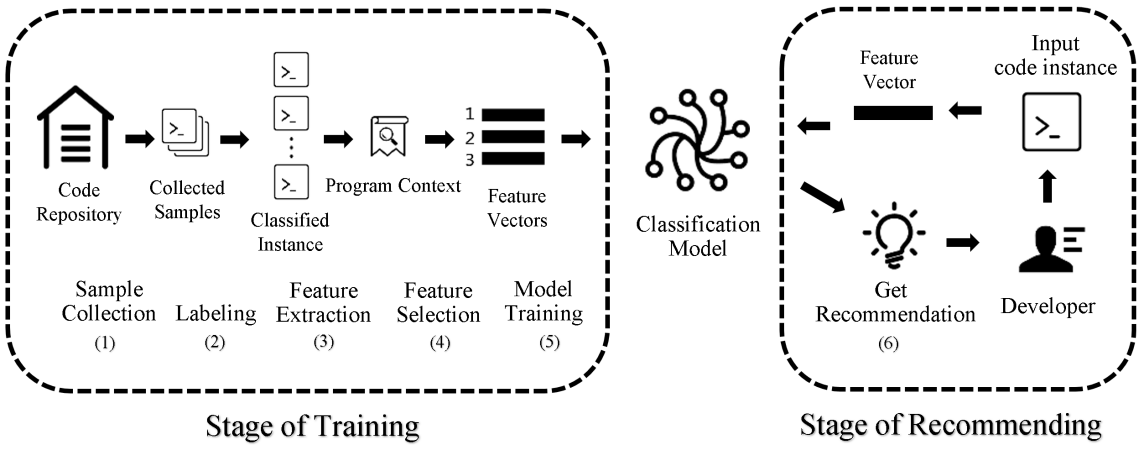

Fig. 2. The workflow of EH-Recommender approach

including the names of the package and classes. (3) Functional context, which represents the functionality of the current code fragments, including the API sequences of code fragments, the name and comment of the current method.

Feature selection: The features extracted from code fragments including the comments of exception and method. This might lead to too many features. For example, the number of features extracted from source code of Hadoop, Hive and HBase exceed 10,000, even if we set the threshold of word frequency to 5. Too many features will cost large amount of time during the model training. Besides, a large part of these features contribute almost nothing in decision-making of exception handling. To solve this issue, we adopt a well-known feature selection method, the Information Gain [12], to eliminate most of irrelevant features. For each sample, we set different information gain to reduce the features to around 250-350. By this way we achieve the best balance of prediction accuracy and training efficiency.

Model training: After obtaining the program context features, we transform these features into corresponding feature vectors, which are divided into the training set and the test sets. For the imbalance nature of different exception handling strategies in source code, we adopted a well-known SMOTE [13] algorithm to address this issue. After all the above processes, we adopt a classification algorithm, such as SVM, Naive Bayes, Random Forest and CNN (Convolution Neural Network), to learn the data and construct the classification model.

Get recommendation: At the stage of recommendation, the trained classification model is utilized to recommend exception handling strategies based on the program context. The target code fragments along with their context are transformed into feature vectors as the same way as the training stage. Then the featured data are input into the classification model, and the model will tell which exception handling strategy is the most appropriate one.

B. Anti-patterns and Noise Data

The quality of training data is crucial to machine learning techniques to get an optimized classification model. However, empirical studies [7] shows that even the most experienced developers can make mistakes in dealing with exceptions. These improper handling code are recognized as noise data which should be filtered out from the training set.

We recognize improper exception handling code according to well-accepted exception handling anti-patterns [7], [14], [8]. In our experiment, we found there are 5 anti-patterns occur frequently in projects, as shown in Table II.

(1) Catching Generic Exceptions. Instead of catching specific exceptions, developers use a single catch block to collect all exceptions. As a result, it may be difficult to determine why the exception was thrown.

(*2) Ignoring exceptions*. The empty blocks defeat the purpose of the exception handling mechanism for it prevents programs from recovering from exceptional execution paths.

(3) Logging and returning null. Developers return null after a log statement in the catch block. This case is useless to handle the exceptions in programs. In the meanwhile, program may run into unintended states. Besides, it will make troubleshooting more difficult.

(4) Catch unchecked exception. Unchecked exception errors are caused by programming errors. However, if we check the proper condition, it can be avoided. Therefore, this kind of exception should not be caught.

(5) Logging and re-throwing. It is not encouraged to catch and throw the original exception in catch block. If you catch the exception but do nothing, this code block is useless. Some developers prefer logging the error and then throwing it. It is also a bad practice of exception handling, because the same exception may produce multiple log records.

Besides the instances conforming to anti-patterns, we also filter out a part of exception handling instances which choosing the wrong handling strategies. We adopt an advanced K-NN algorithm to detect such noise data. Traditional K-NN algorithm detect noise by judging the number of other kinds

TABLE II
ANTI-PATTERNS AGAINST EXCEPTION HANDLING

Anti-pattern	example
Catching Generic Exceptions[7]	catch (Throwable e) { ... }
Ignoring exceptions[7]	catch (IOException e) { //empty block }
Logging and returning null[14]	catch (IOException e) { log.error("error"); return null; }
Catching unchecked exception[7]	catch (ArithmeticException e) { ... }
Logging and re-throwing[8]	catch (IOException e) { log.error("error"); throw e;}

```
import org.apache.hadoop.hbase;
Public class RegionTransition {
 /* @param data Serialized date to parse.
  * @return A RegionTransition instance made of the passed <code>data</code>
  * @throws DeserializationException */
public static RegionTransition parseFrom(final byte[] data) throws DeserializationException
{
    ProtobufUtil.expectPBMagicPrefix(data);
    try {
       int prefixLen = ProtobufUtil.lengthOfPBMagic();
       ZooKeeperProtos.RegionTransition rt=ZooKeeperProtos.RegionTransition.newBuilder()
                                        .mergeFrom(data,prefixLen,data.length-prefixLen)
                                        .build();
       return new RegionTransition(rt);
    } catch (InvalidProtocolBufferException e) {
       throw new DeserializationException(e);
    }
  }
}
```

```
/ * Thrown when a protocal message being parsed is invalid in some way,
  * e.g. it contains a malformed variant or a negative byte length.
  */
public class InvalidProtocolBufferException extends IOException {  …
```

```
Label: Wrap&Rethrow
                        Exception Context
ExceptionType: InvalidProtocolBufferException IOException
ExceptionName: invalid protocol Buffer
ExceptionComment: thrown:1 protocol:1 message:1 parsed:1 invalid:1 contains:1
malformed:1 variant:1 negative:1 byte:1 length:1
                        Architectual Context
ClassName: region transition
PackageName: org apache hadoop hbase
ExceptionalMethodInvocation: google protobuf abstract message builder merge from
                        Functional Context
MethodDeclaration: parse from
APISequence: lengthOfPBMagic newBuilder mergeFrom build
MethodComment: data:3 serialized:1 parse:1 region:1 transition:1 instance:1
made:1 passed:1 byte:1 array:1
```

Fig. 3. An example of features extracted from a HBase code fragment

of data which are most closely neighboring the target data. The advancement in our method is that we take the distances between the target code instance and its 5 nearest neighbors' (5-NN) as the weight when judging whether the target data is noise data. This kind of method can avoid the negative affects resulted from unbalance of sample number in each category to some extent.

C. Program Context and Feature Extraction

Figuring out what factors in the program context dominate the choice of exception handling strategy is critical to our research. The features selected can strongly influence the efficiency of model training and the accuracy of the exception handling strategies recommendation.

The exception handling decision-making largely depends on the program context of the exceptional code. We need to ensure that the extracted features have direct or indirect relationship with the reason why develops choose a specific strategy. Based on the above considerations, we construct a feature set coming from three kinds of program context: exceptional context, architectural context and functional context.

As shown in Fig.3, a code fragment raises an InvalidProtocolBufferException exception in method parseFrom of class RegionTransition. The features extracted are divided into three groups to represent three types of context.

(1) *Exceptional context* is the context of the exception itself. In order to properly handling an exception, developers need to analyze what type of error cause this exception. Similar errors are always handled in similar manners. Such information always hides in the exception type, name and comments. Therefore, the features of exceptional context include three parts: (A) The exception type, including the class name and its super classes; (B) The exception name, including a list of words by splitting the exception name; (C) The exception comments, including a list of words by tokenizing the exception comments.

(2) *Architectural context* is the context indicating the logical level where current code lies, and the source method of the exception. Developers analyze the logical layer based on the package name and class name. In order to get generic features for matching similar logical layer, the class name and package name are split into a set of words, which can be represented into feature vectors by word embedding.

Exceptional method invocation refers to the statement that may trigger an exception, which indicates the source method of exception in the call stack. This feature tells if the exception comes from a system API, a well-known third-party library, or an user-defined method.

(3) *Functional context* is the context representing the functionality of the code fragment. Different functionality might have different exception handling strategies. Some functional tasks can tolerate errors while others not.

The functional context comes from the API sequence, name and comment of the current method. The method name is split and the comments are tokenized into a set of words, and the API sequence is a list of API invoked in current method. The method name and comment tell what job is done by this method, and the API sequence tells how this method doing the job. These two features represent the functionality of the code fragment.

For the textual features, we pre-process them by stemming, stop words removing, tokenization, TF-IDF weighting technique [15], [16], [17], and convert text to vectors through word embedding techniques.

Fig.3 shows an example of the context features we extracted from an exception handling code instance. It is from HBase source code, the file path of this code snippet is: "org.apache.hadoop.hbase.RegionTransition.java".

During traversing the source code, we find that invoking the method "mergeFrom" might trigger InvalidProtocolBufferException, so we analyze its containing method "parseFrom" and find this exception is caught and re-thrown as a new DeserializationException. So, we label this exception instance as WRAP&RETHROW.

IV. Evaluation

A. Corpus

TABLE III
THE CORPUS OF OUR EXPERIMENTS

Projects	Code Fragments	THROW	LOG& IGNORE	HANDLE	WRAP& RETHROW
Eclipse Che	7603	5915 (77.8%)	612 (8.0%)	263 (3.4%)	813 (10.7%)
Consulo IDE	5784	3546 (61.3%)	992 (17.2%)	515 (8.9%)	731 (12.6%)
Directory Server	3843	2589 (67.4%)	379 (9.8%)	294 (7.6%)	581 (15.1%)
Hadoop	41512	31433 (75.7%)	3003 (7.3%)	3010 (7.3%)	4066 (9.8%)
Hama	534	428 (80.1%)	44 (8.2%)	13 (2.4%)	39 (7.3%)
HBase	11348	8499 (74.9%)	736 (6.5%)	714 (6.3%)	1399 (12.3%)
Hive	13215	9265 (70.1%)	750 (5.7%)	545 (4.12%)	2655 (20.1%)
Tomcat	3966	2647 (66.7%)	539 (13.6%)	421 (10.6%)	359 (9.1%)
Activiti	329	66 (20.1%)	69 (20.9%)	27 (8.2%)	167 (50.7%)
Zeppelin	3350	2455 (73.3%)	544 (16.2%)	82 (2.4%)	269 (8.0%)

All exception handling instances are collected from 10 rigorously selected open source Java projects, and we conduct an experiment to evaluate our approach. Due to the lack of "ground truth" on what is optimal exception handling, while the recommendation quality still needs to guaranteed, our selection of experimental projects strictly in accordance with the following basic principles: 1) The development cycle of the target projects is long enough to reach high reliability; 2) The capabilities of the development team have been recognized by the industry; 3) The target open source software system is widely used in the production environment, and the code quality is widely praised. The projects we chose as experimental data sets are shown in Table III.

According to these criteria, 10 projects are selected from open source community.

- *Eclipse Che*: The next generation IDE and developer workspace server. It contains 780K lines of code and 12635 exception handling instances.
- *Consulo IDE*: Cross-platform IDE based on IntelliJ. It contains 1.32M lines of code and 9702 exception handling instances.
- *Directory Server*: Apache LDAP directory server. It contains 440K lines of code and 8273 exception handling instances.
- *Hadoop*: Distributed System Infrastructure. It contains 2.7M lines of code and 88149 exception handling instances.
- *Hama*: A framework for scientific computation. It contains 54.7K lines of code and 761 exception handling instances.
- *HBase*: A high-reliability distributed storage system. It contains 784K lines of code and 23566 exception handling instances.
- *Hive*: A data warehouse tool for statistical analysis. It contains 1.63M lines of code and 52613 exception handling instances.
- *Tomcat*: A commonly used lightweight web application server. It contains 1.57M lines of code and 6874 exception handling instances.
- Activiti: A Business Process Management (BPM) and workflow system. It contains 324K lines of code and 329 exception handling instances.
- Zeppelin: A web-based notebook that enables interactive data analytics. It contains 152K lines of code and 3350 exception handling instances.

B. Metrics

We evaluate EH-Recommender using a metric macro-f1, due to the recommendation of exception handling strategies is a multi-classification prediction problem. A macro-f1 score ranges from 0 to 1, it can strengthen the effect of small sample classes on overall accuracy, which exactly applies to our need, it is calculated as follows:

$$Macro\text{-}P = \frac{1}{n}\sum_{i=1}^{n} P_i \tag{1}$$

$$Macro\text{-}R = \frac{1}{n}\sum_{i=1}^{n} R_i \tag{2}$$

$$Macro\text{-}F1 = \frac{2 \times Macro\text{-}P \times Macro\text{-}R}{Macro\text{-}P + Macro\text{-}R} \tag{3}$$

Corresponding to our classification scenario, n takes a value of 4 since we classify the exception handling strategies into 4 classes. The calculation of $Macro\text{-}F1$ takes N-classification processing as n times two-classification, it calculates the Precision and Recall of each class under two-classifications respectively and perform an arithmetic average of them to obtain the macro average precision score $Macro\text{-}P$ and the macro average recall score $Macro\text{-}R$. Finally, the macro-f1 score $Macro\text{-}F1$ is obtained by Formula 3.

The macro-f1 score weights equally all the classes, regardless the amount of data instances. Therefore, it can avoid the impact of unbalanced data on the predicted results. Take the sample of Hive for example, the total number of data instances reaches 13,215 while the number of handled exceptions is only 545 (4.12%), and the second least instances of Log&Ignore are 750 (5.7%). In the situation all instances with the label Handle and Log&Ignore are wrong recommended, the accuracy can achieve 90.18% (assume other two classes are all correctly recommended), but its $Macro\text{-}F1$ is as low as 0.476.

C. Experiment

Preprocessing : The source code of most projects are obtained from open source repository GitHub. We employ eclipse JDT to analyze the Java code and extract the code instances related to exception handling, and the program context features are obtained from the AST (Abstract Syntax Tree) and the Java Model. During analyzing the collected data instances, we find a certain number of duplicate instances. We checked the source code of these instances and find these duplicate instances are caused by invoking the same method (which trigger exceptions) more than one time. We removed these duplicated instances before we training the models [18]. After obtaining these original samples, we conduct the feature selection and noise data removal process using Python with some third-party libraries, like NumPy [19].

training set and test set: To simulate the real scenario when we apply the EH-Recommender in real programming practice, i.e., the EH-Recommender should learn exception handling practices from high-quality projects and existing high-quality code of the developing project, we divide the exception handling instances in each project into training set and test set by the ratio of 8:2. All the training set of these projects are mixed together and an integrated training set, which are used to train the prediction model. After the training, we use the trained model to evaluate the test set of each project and calculate the metrics of each project.

Training: We train and evaluate the traditional machine learning models (including Naive Bayes, Decision Tree(J48), Random Forest) with the data mining software WEKA [20]. Using WEKA can easily compare the effectiveness of different machine learning models.

The CNN model is trained and evaluated based on TensorFlow [21]. The model contains 3 layers including 1 convolution layer and 2 fully-connected layers. Our vocabulary size is 5000, and we define the dimension of word vector to be 64. Based on the word indices, we can obtain the corresponding embedding vector of each single word. Then, the vectorized exception handling instances can be loaded into the convolution layer. The convolution layer has 256 filters, with the kernel width setting to 7, and followed by a max pooling layer. The first fully connected layer contains 128 hidden units with the dropout setting to 50% keep probability, and using the ReLu activation function. The second fully connected layer is used for classification task, it has 4 hidden units and uses the Softmax activation. During training, the model is trained for 30 epochs, and in each epoch we apply all of the training set to train our model in batches, each batch includes 64 samples. To evaluate the performance, we output the loss and accuracy every 100 batches and write into tensor board scalar every 10 batches. To reduce the loss, we use the Adagrad Optimizer in our method.

The metric baselines: To evaluate our prediction model, we adopt two models as experiment baselines. These two models simulate the scenario where the programmers with no development experience dealt with exceptions and calculate their macro-f1 scores for comparison.

1) We assume the probability is equal that each data instance to be randomly predicted into one of the 4 strategies, in which case the $Macro\text{-}F1$ value is 0.28.

2) We assume that all exception instances are directly thrown from the method signature, based on the observation that in the 10 target projects, the instance ratio of throw exceptions to catch exceptions (including HANDLE, LOG&IGNORE and WRAP&RETHROW three classes) reaches averagely 7:3, which indicate that the exceptions triggers by method invocations are more likely to be thrown than caught. In this case, the $Macro\text{-}F1$ value is 0.18.

Experiment result: Comparing with the above two baselines, the promising result of EH-Recommender indicates its high effectiveness and availability in practical development, where the $Macro\text{-}F1$ value of each test set ranging from 0.718 to 0.967. And the specific result (of CNN model) of each exception handling strategy is shown in Fig. 4. The values shown in Fig. 4 are the F1-score of each class, which consider both the Precision and Recall of the classification. Fig. 4 shows, the EH-Recommender reaches a high-level accuracy in every class of each data set. Generally, the F1-score of category THROW is the best of them all, it is as expected because the instance number of this class far exceeds the others. Besides, the lower accuracy of recommending other strategies is caused by some objective factors, the field study of Cabral et al. [22] on Java and .Net projects points out that different projects significantly differ in the subsequent processing steps after catching exceptions. Since the projects used in our experiment have a large variety in terms of project type and scale, the results in Fig. 4 are still very promising.

We also notice that in the data sets of Hadoop, HBase and Hive, the proportion of each class is nearly the same, and the distribution of the prediction effects of these projects are also similar; the similarity of the same classification effect distribution also appears in the data sets of Che and Consulo. This phenomenon leads us to think about whether the source code with the same project type (or have potential relation) have similar preference in choosing exception handling strategies.

The comparison of CNN model with other machine learning models is posted in Table IV. The predictive accuracy of these models in Table IV is represented by the macro-f1 score. Between the machine learning models, the lowest accuracy appears in Naive Bayes model, it bases on the probability theory, a possible explanation of its low accuracy is that there exists a certain correlation between context features, which is not suitable to be handled with Naive Bayes. SVM can achieve excellent accuracy in small sample classification, however, because of our training set, the accuracy of SVM is lower than that of Decision Tree. The predictive accuracy of other three machine learning models is too large, the accuracy of SVM is not satisfying enough, and it is also the most time-consuming model. Decision Tree and Random Forest have similar predictive accuracy, but they still can not compete with the CNN classification model, whose $Macro\text{-}F1$ value are averagely 0.810, it is a very promising result in multiple classification problem.

TABLE IV
THE MACRO-F1 SCORE OF DIFFERENT CLASSIFICATION ALGORITHMS

Models	Eclipse Che	Consulo IDE	Directory server	Hadoop	Hama	HBase	Hive	Tomcat	Activiti	Zeppelin
Naive Bayes	0.304	0.210	0.471	0.157	0.266	0.165	0.252	0.282	0.393	0.401
SVM	0.549	0.507	0.735	0.511	0.378	0.579	0.687	0.663	0.420	0.637
Decision Tree	0.672	0.655	0.863	0.781	0.438	0.757	0.719	0.697	0.509	0.697
Random Forest	0.691	0.705	0.873	0.806	0.447	0.768	0.738	0.701	0.553	0.736
CNN	0.730	0.718	0.88	0.838	0.968	0.813	0.825	0.780	0.740	0.817

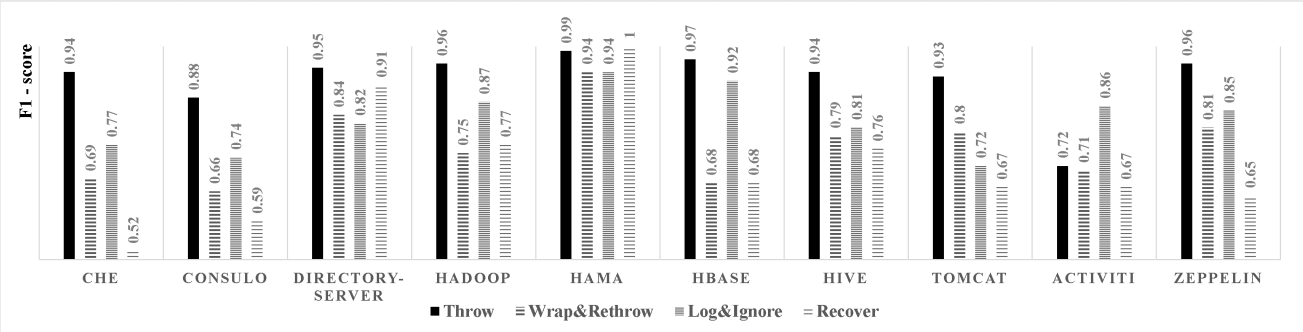

Fig. 4. The CNN result of EH-Recommender approach

V. THREATS TO VALIDITY

In this section, we present four major threats to the validity of our work.

(1) The code quality of corpus. Our approach tries to learn the good exception handling practices from high-quality projects and code of well-skilled programmers. However, there is no absolute standard to evaluate the quality of the code. At the same time, the experience of exception handling may be threatened by the limited diversity of projects. To mitigate this threat, we tested 10 projects with more than 90,000 data instances. The type and scale vary a lot between our selected projects, also we have strict restrictions on the quality of projects. Most of the projects we eventually chose were top projects under Apache. These projects have received attention and recognition from developers all over the world and have been widely used. In addition, the results of noise detection can also prove the effectiveness of our classification strategy and the high-quality of the selected code from another perspective. In the future work, we can further improve the application scenarios of our methods based on more projects, which can further reduce the threat of diversity.

(2) The features of program context. The selection of features are based on the program context which influences the exception handling. There are no commonly accepted rules to tell how much context contributes to the exception handling decisions. The features in this paper cover the exceptional context, which represents the type and the cause of the error, the architectural context, which represents the logical layer and the source exceptional method, and the functional context, which represents the functionality of the code. These contexts are closely related to the exception handling decision, but can not cover all the information related to exception handling decision-making.

(3) The generality of features. Most time, the exception type, package name, method name and APIs are all program specific. We need to generalize them to alleviate the over-fitting problem. Therefore, we introduce the comments of these exceptions and methods into features, and also split method names into words, and adopt the word embedding technique to represent the general semantics of such context. In fact, to some extent, the features of the logical layer are still not so generic, which makes cross-project recommendation hard to get high precision. The more general features of the logical layer need high-level design information, which implicitly exists in the code, but is difficult to extract and represent. This is our next important work in the future.

(4) Data imbalance of different strategies. The number of code fragments of different strategies are quite unbalanced. The strategy THROW has most of the data, because an exception always goes through several methods before reaching a place to handle it. We under-sample the data of this category by removing some code fragments which have similar features. And also, we adopt the well-known SMOTE oversampling technique to generate more samples for other categories if they have very few samples.

As far as we know, this paper is the first work on exception handling decision-making based on machine learning. Therefore, it is still far from being a mature and practicable approach. Some limitations of this work are as follows:

(1) Projects always differ significantly in the subsequent processing after catching an exception, this has been proved by other studies [22]. Therefore, cross-project recommendation is still a problem. One way to improve predictive accuracy of

catching exceptions is to learn from more diverse projects, and mining their project types and relevance as feature dimensions. Also, we need more generic features for the architectural context, to enable better cross-project feature matching.

(2) Currently, our work mainly focuses on checked exceptions. The RuntimeException and its sub-classes appear frequently in practical development as well as checked exceptions. These exceptions are not caught or thrown explicitly, therefore observing and mining the rules of dealing with RuntimeException usually involve implicit exception flow analysis of related code. So, recommending the RuntimeException handling strategies will be more challenging.

(3) In this paper, we provide 4 generic exception handling strategies for developers to assist them in handling exceptions. This classification approach to exception handling strategies is sufficient for most development scenarios, but it can still be better. For example, in the case of throwing exceptions directly, it can further analyze where the exception is finally handled, knowing this can largely prompt our recommendation; in the case of catching exceptions, there exists some special exception handling modes, however these data instances are rare so it is difficulty to learning from these exception handling patterns. We can try to capture these specific exception handling patterns with the emerging transfer learning and few-shot learning techniques.

VI. Related Works

Exception handling is an active research area for many years. Considerable empirical studies [23], [11], [24], [6], [4], [25] and user surveys [7], [26], [27], [8] have been conducted to analyze the exception handling practice among different programming languages, these studies reveal the developer's exception handling behavior and it inspired our work largely. Many researchers studied the relationship between exception handling code and software robustness [28], Osman et al. [29] and Cacho et al. [25] evaluated how changes in exceptional code can impact system robustness. Marinescu et al. [4] suggest that classes using exceptions are more complex than those not using exceptions. Moreover, classes that handle exceptions in an improper manner show a higher probability of exhibiting defects than classes which handle them properly. In the meantime, experiments [9] have proven that the exception handling code itself may also cause errors, such errors were called EH-bugs (exception handling bugs), and the exception handling defect density of exception handling constructs is approximately three times higher than overall defect density.

The inherent complexity in exception handling is one of the reasons why many developers end up oversimplifying [23], [3], [30] or even neglecting exception handling [31], [8] in their programs. Therefore, some works paid attention to analyzing the exception handling code, and hoped to aid developers in understanding the global exceptions flows of the program [24], [32], [33], [34].

Recent years, some new approaches are presented, to mine or learn exception handling patterns from code repositories, or to recommend exception handling code based on context similarity.

Thummalapenta et al. [11] develop a novel approach that mines exception-handling rules as sequence association rules of the form "$(FC_c^1...FC_c^n)\bigwedge FC_a \Rightarrow (FC_e^1...FC_e^m)$, which is required to characterize common exception-handling rules. They present a technique for learning specialized instances of specifications for exceptional code paths, and directly mine sequential patterns by using closed frequent sequential pattern mining. However, these exception handling patterns are based on the existing exception handling code, and how the exception is ultimately handled this processing model is based on the existing exception handling code to analyze how the exceptions are eventually handled. But our work is to propose the recommendation of exception handling during development.

Rahman et al. [35] and Barbosa et al. [36] proposed a method based on code context similarity, which can search for a similar exception handling code example in the open source code repository. However, this method focuses mainly on the subsequent processing after catching an exception, it ignores the decision that whether the exception should be caught or be thrown directly. Meanwhile, this method is evaluated on a small data set, which cannot provide the developer a straightforward advice for handling exceptions. Our approach is to take advantage of the existing good exception handling experience to provide developers with a direct and operational handling recommendation. And based on the classification method of exception handling strategies, EH-Recommender can help developers to handle exception more efficiently.

VII. Conclusion

Exception handling is a development task which is of significant importance to the code robustness and the software reliability. However, how to handle exceptions properly is still a challenge for many developers, due to the lack of specific rules and effective tools for assistance. To bridge this gap, in this paper, we propose the approach named EH-Recommender. The basic idea is to learn the good exception handling experience from existing high-quality open source projects and well-skilled developers and leverage the learned model to recommend appropriate exception handling strategies for developers. The experiment achieves promising recommendation accuracy, which indicates that the practical value of EH-Recommender in helping developers improve the quality of exception handling code. In the next step, we are preparing to improve the quality of our recommendation by extracting more general high-level architectural features, and exploring the transfer learning and few-shot learning technique, to improve the performance of cross-project recommendation.

Acknowledgment

We thank Prof. Jin Liu and Prof. Xiaoyuan Xie for their suggestions; we thank Tengda Tang and Xiaohui Hu for their useful help in experiments. This work is supported by the National Key Research and Development Program of China

(No. 2016YFC1202204), the National Science Foundation of China (Nos. 61672392, 61373038, 61502345, and 61872273), and the Technological Innovation Projects of Hubei Province (No. 2017AAA125)

REFERENCES

[1] J. J. Bloch, *Effective Java, 2nd Edition*. Addison-Wesley, 2008. [Online]. Available: http://www.worldcat.org/oclc/255160742

[2] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Commun. ACM*, vol. 18, no. 12, pp. 683–696, 1975. [Online]. Available: http://doi.acm.org/10.1145/361227.361230

[3] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, 2009, pp. 307–318. [Online]. Available: https://doi.org/10.1109/ASE.2009.94

[4] C. Marinescu, "Should we beware the exceptions? an empirical study on the eclipse project," in *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, 2013, pp. 250–257.

[5] R. Coelho, L. Almeida, G. Gousios, A. van Deursen, and C. Treude, "Exception handling bug hazards in android - results from a mining study and an exploratory survey," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1264–1304, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9443-7

[6] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015. [Online]. Available: https://doi.org/10.1016/j.jss.2015.04.066

[7] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in java?" in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 516–519. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903500

[8] H. Shah, C. Görg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Trans. Software Eng.*, vol. 36, no. 2, pp. 150–161, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2010.7

[9] P. Sawadpong, E. B. Allen, and B. J. Williams, "Exception handling defects: An empirical study," in *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, 2012, pp. 90–97.

[10] E. A. Barbosa and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," *IEEE Transactions on Software Engineering*, 2017.

[11] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 496–506.

[12] C. C. Aggarwal and C. Zhai, "A survey of text classification algorithms," in *Mining Text Data*, C. C. Aggarwal and C. Zhai, Eds. Springer, 2012, pp. 163–222. [Online]. Available: https://doi.org/10.1007/978-1-4614-3223-4_6

[13] K. W. Bowyer, N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *CoRR*, vol. abs/1106.1813, 2011. [Online]. Available: http://arxiv.org/abs/1106.1813

[14] G. B. de Pádua and W. Shang, "Studying the prevalence of exception handling anti-patterns," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 328–331. [Online]. Available: https://doi.org/10.1109/ICPC.2017.1

[15] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 415–425. [Online]. Available: https://doi.org/10.1109/ICSE.2015.60

[16] B. Bassett and N. A. Kraft, "Structural information based term weighting in text retrieval for feature location," in *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 2013, pp. 133–141. [Online]. Available: https://doi.org/10.1109/ICPC.2013.6613841

[17] W. Zheng, Q. Zhang, and M. R. Lyu, "Cross-library API recommendation using web search engines," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 480–483. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025197

[18] J. Xuan, Y. Gu, Z. Ren, X. Jia, and Q. Fan, "Genetic configuration sampling: learning a sampling strategy for fault detection of configurable systems," in *Proceedings of the 5th International Workshop on Genetic Improvement, GI@GECCO 2018, Kyoto, Japan, July 15-19, 2018*, 2018, pp. 1624–1631.

[19] W. McKinney, *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.", 2012.

[20] M. A. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[22] B. Cabral and P. Marques, "Exception handling: A field study in java and .net," in *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, 2007, pp. 151–175.

[23] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in java projects: an empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 500–503.

[24] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of java libraries: an empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 212–222.

[25] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. García, E. A. Barbosa, and A. Garcia, "Trading robustness for maintainability: an empirical study of evolving c# programs," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 584–595.

[26] D. Reimer and H. Srinivasan, "Analyzing exception usage in large java applications," in *Proceedings of ECOOPï¿œï¿œ2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003, pp. 10–18.

[27] M. Monperrus, M. G. de Montauzan, B. Cornu, R. Marvie, and R. Rouvoy, "Challenging analytical knowledge on exception-handling: An empirical study of 32 java software packages," Ph.D. dissertation, Laboratoire d'Informatique Fondamentale de Lille, 2014.

[28] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in java server applications," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 230–239. [Online]. Available: https://doi.org/10.1109/ICSE.2007.35

[29] H. Osman, A. Chis, C. Corrodi, M. Ghafari, and O. Nierstrasz, "Exception evolution in long-lived java systems," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 302–311.

[30] X. Zhang, Y. Chen, Y. Gu, W. Zou, X. Xie, X. Jia, and J. Xuan, "How do multiple pull requests change the same code: A study of competing pull requests in github," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain. September 23-29, 2018, to appear*, 2018.

[31] H. Shah, C. Görg, and M. J. Harrold, "Visualization of exception handling constructs to support program understanding," in *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008*, 2008, pp. 19–28. [Online]. Available: http://doi.acm.org/10.1145/1409720.1409724

[32] B. Cornu, L. Seinturier, and M. Monperrus, "Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions," *Information & Software Technology*, vol. 57, pp. 66–76, 2015. [Online]. Available: https://doi.org/10.1016/j.infsof.2014.08.004

[33] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM*

Trans. Softw. Eng. Methodol., vol. 12, no. 2, pp. 191–221, 2003. [Online]. Available: http://doi.acm.org/10.1145/941566.941569

[34] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, "B-refactoring: Automatic test code refactoring to improve dynamic analysis," *Information & Software Technology*, vol. 76, pp. 65–80, 2016.

[35] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, 2014, pp. 285–294. [Online]. Available: https://doi.org/10.1109/SCAM.2014.15

[36] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *26th Brazilian Symposium on Software Engineering, SBES 2012, Natal, Brazil, September 23-28, 2012*, 2012, pp. 171–180. [Online]. Available: https://doi.org/10.1109/SBES.2012.22