Detecting Anti-Patterns in Java Exception Handling using Static Code Analysis

*https://github.com/LeoWangCoder/AntiPattern

Bo Yang
40122758
Concordia University
Montreal

Junjie Li
40120408
Concordia University
Montreal

Zehao Wang
40121380
Concordia University
Montreal

Yuxuan Luan
40112891
Concordia University
Montreal

Abstract—**Java provides exception handling mechanism to enhance the practice of software reliability, comprehension, and maintenance. However, prior studies suggested some anti-patterns and showed some of them have a non-trivial prevalence. Static Analysis is an effective and efficient way to detect problems without execution. It can detect some potential security violations, run-time errors, logical violations and some more. In this study, we developed a static java code analysis Eclipse-plugin to detect three exception handling anti-patterns - Nested-Try, Destructive Wrapping and Over-Catch using Eclipse Core and Eclipse Java Development Tools (JDT). In this way, developers can detect these anti-patterns by simply clicking the plugin.**

I. Introduction

Java provides exception handling mechanism to enhance the practice of software reliability, comprehension, and maintenance. Prior studies [1], [2] suggested some anti-patterns and showed some of them have a non-trivial prevalence. Static Code Analysis is a method of analyzing the source code of programs without running them. Eclipse Core [3] component provides API to access projects' source code and their dependent libraries in the workspace. Eclipse JDT [4] contributes a set of plugins that add the capabilities of a full-featured Java IDE to the Eclipse platform. In our study, we developed an Eclipse-plugin using Eclipse JDT and Core, which could detect three types of exception handling anti-patterns:

1) Nested Try: This anti-pattern is caused by one or multiple try statements (including try-with-resources statement) represent in another try catch block. The misuse of this anti-pattern is not readable and maybe even confusing to some developers and it is more prone to error.
2) Destructive Wrapping: This anti-pattern is caused when an exception is caught, it does not throw a complete exception information in catch blocks. The misuse of this anti-pattern may cause of the destroy the stack trace of the original exception.
3) Over-Catch: In this anti-pattern, a higher level exception is used to catch multiple different lower-level exceptions. *D. Yuan et al.* [2] found that 8% of the catastrophic failures were caused by developers prematurely aborting the entire cluster on a non-fatal exception.

II. Methodology

We leverage Eclipse Core component to access the projects' source code and their dependent libraries in the workspace. We utilize ASTParser of JDT to parse the source code and binary files, and use ASTVisitor of JDT to traverse the AST and find the target patterns.

a) Nested-Try: First, we use a visitor to traverse the class's AST. After identifying a TryStatement, we created a new visitor to traverse the body of TryStatement. In the body of the original TryStatement, if there exists TryStatement, then it is a Nested-Try case.

b) Destructive Wrapping: To detect this anti-pattern, we need to collect the exceptions in catch clause first, then analyze the exceptions in throw statements. In a catch block, the variable names of two exceptions will be compared. if they are the same, this catch block will be considered as a right handling. Otherwise, an anti-pattern.

c) Over-Catch: For this anti-pattern, we need to analyze all possible exceptions in the try block of the source code and check if they are caught by their super class. In java, all exceptions are inherited from the java.lang.Exception. Unchecked exceptions are the RuntimeException and its sub-classes. The others are checked exceptions. A checked exception must be either caught or declared in the method. So we can get the checked exceptions from the method binding easily. However, for the unchecked exceptions, we have to analyze the possible exceptions from the call graph. It could have three sources: throw statement in the source code, local java doc and online java doc. In general, we have 3 steps:

1) Identify all checked exceptions in the try block.
2) Identify all unchecked exceptions from the call graph of the method invocations in the try block.
3) Analyze if any exception is caught by its super class.

In the call graph, some methods could throw some exceptions and the callers may handle or may not handle them. We need to decide what exceptions could arrive at the try block finally. We recursively visit the method call (both source code and libraries) until there is no more method call or the method call's compilation unit is unavailable for some reasons (e.g.

TABLE I
PROJECTS INFORMATION

Project	Version	#Throws	#Catch	Method (K)	KLOC
Fastjson	Commit at f164d85	667	1001	18	172
JUnit4	Commit at 435d41f	2344	319	4	31
RefactoringMiner	Commit at 5f13412	176	88	2	25

TABLE II
ANTI PATTERN STATISTICS

Projects	Nested-Try	Destructive Wrapping	Over-Catch
Fastjson	7	14	258
JUnit4	3	15	84
RefactoringMiner	1	0	17
Total	11	29	359

the method is a native method). In each method, we collect all the exceptions from the throw statements, local java docs and online java docs. Also, we retrieve exceptions from the methods it called and check if they are caught. For the thrown exceptions and local java doc, we could resolve bindings and check the inheritance relationship between the thrown exception type and the caught exception type. For the online java doc, we only have the text information. Our strategy is removing the exceptions that are caught (by comparing the qualified name) and thrown (the doc may specify both the checked and unchecked exceptions) from the online java doc exceptions set. If there are still exceptions in online java doc exceptions set and the try block catches some generic exceptions (i.e. java.lang.Throwable, java.lang.Exception, java.lang.RuntimeException), then it could be an over-catch case.

III. EVALUATION

We run our tool on the *fastjson* [5], *JUnit* [6] and *RefactoringMiner* [7]. *Fastjson* is a Java library that can be used to convert Java Objects into their JSON representation. *Junit* is a programmer-oriented testing framework for Java. *RefactoringMiner* is a library/API written in Java that can detect refactorings applied in the history of a Java project. The Table I shows the project information and the Table II shows our result.

A. Nested-Try

The Nested-Try is the least cases among these three anti-patterns. Nested-Try makes the code hard to read, especially for the complex long method.

B. Destructive Wrapping

Destructive Wrapping also has few cases. We found that in most of this anti-pattern cases, developers create a new exception with some custom messages (e.g. Fig. 1). We think in these cases, developers care about the message much more than the stack trace. But still, it is a good practice to keep the original stack trace.

```
catch (IllegalArgumentException e) {
   throw new RuntimeException("unexpected:
       argument length is checked");
} catch (IllegalAccessException e) {
   throw new RuntimeException("unexpected:
       getMethods returned an inaccessible
       method");
}
```

Fig. 1. Destructive Wrapping case in `JUnit4`

```
public Object decode(String text){
   try {
      return mapper.readTree(text);
   }
   catch (Exception e) {
      throw new
          RuntimeException(e.getMessage(),e);
   }
}
public final Object decodeObject(String
    text){
   try {
      return
          (ObjectNode)mapper.readTree(text);
   }
   catch (Exception e) {
      throw new
          RuntimeException(e.getMessage(),e);
   }
}
```

Fig. 2. Similar methods containing over-catch in `Fastjson`. There are more than 10 methods similar to these 2.

C. Over-Catch

Over-Catch is the most common anti-pattern in our experiment. We found some interesting things. One is that some of the anti-pattern methods are very similar, this could be caused by code clone (e.g. Fig. 2). Also, there are many cases are from test code. We found that the Over-Catch have different behaviors between normal code and test code. In test code, most cases are just expecting their code could cause some exceptions but do not care the exact exception type (e.g. Fig. 3). Some cases catch Throwable and then check if the caught exception is the same type with the oracle (e.g. Fig.4). We argue that the Over-Catch anti-pattern for the test code is not the same harmful as the anti-pattern for the normal code. Also, junit4 has some special cases for the Over-Catch. They catch all the AssertionError and then throw a custom Failure type. For example in Fig. 5, it compares 2 arrays and if there is any assertion error, they want to show it is an ArraryComparisonFailure rather than the exact assertion error type.

IV. CONCLUSION AND FUTURE WORK

In this study, we introduced our method on detecting 3 common anti-patterns in java exception handling practice using

```java
} catch (Exception ex) {
   error=ex;
}
assertNotNull(error);
```

Fig. 3. Test code in `Fastjson`, which only expect it causes an exception but does not cares about the exact type. There are many similar cases.

```java
catch (Throwable ex) {
   Assert.assertEquals(ex.getCause()
       instanceof ClassCastException,false);
}
```

Fig. 4. Test code in `Fastjson`, which catches a general type and then check the exception type.

static code analysis and evaluated our tool on three open source java projects. From the evaluation, we found that the Over-Catch anti-pattern have different behaviors between normal code and test code and may also have different impacts on the code quality. We suggest future study about exception handling anti-pattern separate the normal code and test code and also call for future work for analyzing the different impacts of exception handling anti-patterns on normal code and test code. Among these three anti-patterns, the first two, Nested-Try and Destructive Wrapping are comparable easy to detect and our tool could finish the detection in a very short time. However, for the Over-Catch, as it will build the full call graph, it could cost a very long time even for a very small project (if it depends on some complex libraries). Also, for the online java doc, as we only have text information, we cannot get the inheritance relation between these kinds of exceptions and the caught exceptions, which could produce false positives. In the future, we will perform some comparison experiments:

1) When building call graph, parsing binary files v.s. only parsing source code files
2) When collecting exceptions, with online java doc v.s. without online java doc

For the experiment 1, we expect it will reduce the detecting time significantly. If, in this case, it won't miss many anti-pattern cases, then it would be better if we do not parse the binary files from the dependent libraries. For the experiment 2, first we want to know if retrieving the online java doc costs lots of time and if online java doc contributes many false positives.

```java
try {
   Assert.assertEquals(expected,actual);
} catch (AssertionError e) {
   throw new ArrayComparisonFailure(header,
       e, prefixLength);
}
```

Fig. 5. Test code in `JUnit4`, which catches AssertError and throw a new Failure.

Also, we will evaluate the precision and recall for our tool in the future.

References

[1] G. B. D. Pádua and W. Shang, "Studying the prevalence of exception handling anti-patterns," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 328–331.

[2] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 249–265. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[3] Eclipse, "Eclipse core," https://www.eclipse.org/eclipse/platform-core/.

[4] ——, "Eclipse jdt," https://www.eclipse.org/jdt/.

[5] Alibaba, "fastjson," https://github.com/alibaba/fastjson.

[6] Junit-team, "Junit," https://github.com/junit-team/junit4.

[7] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206