A New Mutation Analysis Method for Testing Java Exception Handling

Abstract

Java exception mechanism can effectively free a program from abnormal exits and help developers locate faults with the exception tracing stacks. It is necessary to verify whether the exception handling constructs are arranged appropriately. There are some approaches have been developed to evaluate the test sets and improve the quality of them, so that they can raise more number of exceptions in programs. Mutation analysis is a practical method to evaluate the quality of test sets. This paper presents some new mutation operators for Java exception handling construct. Moreover, equivalent mutants can be identified with these mutation operators. A case study illustrates the effectiveness of these mutation operators and discusses their characteristic features.

1. Introduction

To avoid semantic violation, linking program error or limitation of system resource, exception handling construct occurs frequently in Java programs. Most developers use the exception handling construct only to satisfy the basic requirements of Java verification. The lack of exception propagation information makes the exception handler more and more complex [1]. However, the property of high reliability always requires a robust program with complete exception handling constructs.

Software testing technique is required to improve the reliability of programs. There is no universal testing theory to guide programmers to select an adequate method which can satisfy different test requirements [2]. Essential for testing the exception handling construct, the structural method, e.g. local search, is widely applied [3]. The whole structure of program is analyzed to retrieve all the possible executing paths. And then, it takes one kind of measure to produce fitness function: the test cases are executed and evaluated one by one, and high quality test cases are selected from a test set.

In such a process, one challenge is that not all exception points can be located. In Java, an exception can be raised implicitly or explicitly. An implicitly raised exception generates form calling Java library routine. It is difficult to draw out the control flow of calling Java library routine [4]. Before the exception block, there may be not arithmetical or logical control statements. So that it is difficult to design suitable fitness functions and to distinguish the exception exit and the common exit. Apparently, either exception exit or common exit shows the end of execution. It is not trivial to choose a path with exception-caused exit and trace the route of a test case.

We believe that mutation analysis adapts to evaluate test sets for exception testing [5,6]. Mutation technique changes the original program to a mutated one with mutation operators [7,8]. Mutation operators are classified to insert different faults. For example, there are particular mutation operators for Java concurrent programs [9]. In mutation testing, we collect the result after the original program runs firstly. Then we run all mutants and collect the results. If the result of mutant is different from the original result, the corresponding test case used is said to kill the mutant. Test set is sufficient if test cases in such set can kill all generated mutants. This test set has strong ability to check the faults hid in the program.

Our method does not determine exception paths explicitly. We attempt to change exception propagation path according to the characteristic of Java exception handling construct. When adopting the mutation testing, we change the exception blocks firstly. Then we compare the mutated exception blocks to make sure they do show different behaviors of exception handling. After effective mutants are selected, JUnit framework is used to carry the results out to see how well the test cases are designed.

This paper is organized as follows: we firstly provide a background of Java exception handling construct in Section 2. According to the structure of exception, we develop several particular mutation operators in Section 3. The key idea is that the operators change the exception propagation path after
one exception is thrown. We also provide some methods to distinguish the equivalent mutants. In Section 4, a case study is designed and implemented to compare the effectiveness of our operators with those in Jumble. We draw a conclusion and discuss the future work in Section 5.

2. Background

This section provides an overview of exception handling construct in Java. Based on the composing of program, semantic exception hierarchy and invoking exception hierarchy are also introduced. We will mutate the original program based on such two hierarchies.

2.1. Exception handling construct

In Java, an exception mechanism is provided to make the software free from uncontrolled states. Programmers must be careful to use the exception because sometimes indeterminism transpires. Differ from other advanced programming languages, a maturity java exception construct includes three kinds of blocks: try block, catch block and finally block. Catch block and finally block are optional. Every try block must have at least catch block or finally block attached. If catch clause is excluded, control flow will transfer out of the try block, the new exception may propagate through the call stack. Finally block is used to free up resource, e.g. closing files or disposing of graphics. Practically, not all exceptions can be caught and disposed once they are raised. There are synchronous exception and asynchronous exception. For asynchronous exception, the exception handler usually handles exception raised from other program segment [10]. Asynchronous exception itself also happened arbitrarily while synchronous exceptions can be checked and handled at specific statements. Further more, synchronous exceptions include two types of exceptions, unchecked exceptions and checked exceptions. Unchecked exceptions derive from errors. For example, as error, runtime exceptions may be generated in JVM itself. It doesn’t require that unchecked exceptions, compile-time errors, are caught and declared in the method. But checked exception must be declared in its signature when the program is compiled.

Many techniques are developed to guarantee the safety of checked exception handling construct.

Because exceptions can be instantiated, as a subclass of Java.lang.Throwable, and explicitly threw using a throw statement, a method is introduced to use control-flow presentation to indicate all throw statements. Through different execution paths, the technique establishes some exception-cover criteria [11]. It needs to design test cases manually for raising all potential exceptions. The manual work is tedious. N. Tracey et al add fitness expression for exception conditions [12]. Then the instrumented program calculates the fitness to determine the test data raising exceptions. An effective exception handling construct is a tool with which developers can make the exception flow as their wish.

In general, the path of exception is difficult to be determined. Random test cases may be better in some cases. We make test cases for exception handling construct to follow some hierarchies. In this way, test cases can be distinctly judged.

2.2. Semantic exception hierarchy and invoking exception hierarchy

Semantic exception hierarchy is composed by original Java API hierarchy and user-define exception hierarchy. In Java API, exceptions are organized by some exception hierarchies, as it is depicted in Figure 1.

![Figure 1. Semantic exception hierarchy in Java API](image)

Users can define and extend existing exception types in Java API to satisfy their needs. Some brand new exception types can be also defined.

Existing analysis methods focus on the execution path. It is complex to determine the execution path with exception handler. We introduce invoking exception hierarchy which organizes the structure of program according to the relationship of different exception handlers.
Invoking exception hierarchy derives from the sequence of call stack. Figure 2 depicts a segment which includes three methods and two levels of calling. Method main is supposed to catch E and call method 1. Method 1 may throw two kinds of exceptions (E1 and E2) when different paths are chosen. Method 1 calls method 2 which will throw out E12. We take E as the root of the whole hierarchy because E is supposed to represent all instances when method main call other methods. E1 and E2 are E’s children because E1 and E2 represent the concrete situations. Method 1 is supposed to catch E1, so E1 represents the parent of E12.

A try block can be inserted into another try block to classify the exceptions. We take the exceptions to be caught inside as the derived type of the outside exceptions. As showed in Figure 3, method main wants to catch E. We take E as the root of the hierarchy tree. A handler which wants to catch E1 nests in method main. E1 is placed as the child of E. The exception from calling method 1 will be placed as the children of E1. The basic consider is to put the catch block higher level.

We put all exception types which appear in the invoked method under the types in the method which invokes. In the same method, exception types in try block are taken as the child of those in catch block. The exception which is explicitly thrown out of try block is parallel with the catch block in the same exception handling construct. The upper method exceptions are the parents of these two kinds of exceptions.

Another problem is that upper method has several coordinate catch blocks to provide meaningful exception messages, as it is showed in Figure 4. Exception types from these blocks will be placed in the same level of the invoking hierarchy. E1, E2 and E are taken as the parent of E1, E2.
In Section 3, we introduce the method with which the mutation operators are applied using these two hierarchies.

Finally, according to the invoking exception hierarchy in Figure 2, we define that E is **invoker type** of E1, E2 and E12. E1 E2 and E12 is **derived type** of E. E is **direct invoker type** of E1 and E2, while E12 is **direct derived type** of E1.

### 3. Mutation operators

There are many testing techniques applied to guarantee high quality programs. Some researchers focus on the structure of source codes - employing custom algorithms and strategies to find potential errors. R. A. DeMillo et al developed mutation testing to reduce human interactions during testing [13]. It produces mutant version of the original program. Through comparing the execution state and result, it determines which mutant has been killed. The killer, some test cases, would be evaluated. Since then, some automated test data generation algorithms for mutation testing have been created - e.g. CBT (constraint-based test data generation) [14] and DDR (Dynamic domain Reduction test data generation) [15]. Per se, mutant testing offers criteria to test programs and test data.

There exist many mutant operators for testing Java programs [16]. Advance programming languages, such as Java, face to complex test circumstances. Mutation operators of Java contain intra-methods level, inter-methods level, intra-classes level and inter-classes level [17]. Exception mutant operator is design over the intra-method and inter-methods level.

One of the main challenges of mutation test is the high expense of computing. Testers need to load all the mutants and collect the results. We compare output of test cases of JUnit to examine whether the mutant is killed. Because the exception unit in JUnit contains no algorism or logical statements, existing mutation operators are not sensitive enough. The new mutation operators have the ability to cause different outcome in JUnit framework.

Another problem is that mutants may have the same behavior as the initial program, i.e. the mutant is an equivalent one. A mutant is equivalent if and only if there is no test case can distinguish the result of mutant from the result of original program. So it should be identified and removed from the sequence to execute.

There are two types of mutation points which can bring in equivalent mutants. The first type is that the mutation point allows at least one mutant operator to be applied. The other one is that one mutant operator introduces several mutants contemporarily [18]. A. Jefferson et al develop method based on CBT to detect equivalent mutants automatically [19]. We distinguish the equivalent mutants with semantic exception hierarchy in Java.

Our mutation operators pay extra concentration on the catch block. As mentioned in Section 2, catch block will retrieve the control if its parameter of the catch block matches the real exception type. It recommends using multiple catch blocks in order to handle specific exceptions. These catch blocks should be arranged from special to common. There are four different situations when try block raise an exception [20]:

1. Catch block handles exception and types out the trace message and avoid exiting abnormally.
2. Catch block handles exception and re-throws the same exception or another exception.
3. Catch block does not handle exception correctly. The exception propagates upwards.
4. Catch block miss the exception. The exception propagates.

Developers are intent to design catch block to match the second and third situation in some cases. When a new exception is thrown, not enough information is collected to handle this exception appropriately. If the associated catch block matches the type of exception, the exception is swallowed. We try to delete the catch block in the method under the root, so that context up the call chain can handle the exception and move the program on. Our mutation operators effectively change the active path of exception.

### 3.1. CBR - Catch Block Replacement

```java
public void methodFirst(String filename)
    throws IllegalArgumentException,
            FileNotFoundException,
            IOException
{
    … …
    try{
        in= new FileInputStream(filename);
        methodSecond(in);
        //methodSecond may throw
        //IllegalArgumentException or IOException
    }

Original:
Δ catch(FileNotFoundException fnfe){… …}
Mutant1(replace with direct a derived type):
Δ catch(IOException ioe){… …}
Mutant2(replace with another derived type):
Δ catch(IllegalArgumentException iae){… …}
…
```
Using CBR mutation operator, the level of the original method should be determined firstly. Then we get the direct invoker type and direct derived type of original catch parameter type. If the original method is root node in the invoking exception hierarchy, we replace the parameter type with direct derived type to generate some mutants. If original method is leaf node in the invoking exception hierarchy, we replace the parameter type with direct invoker type. And if original method is middle of the hierarchy, we uses direct derived type and direct invoker type in the same time.

The semantic exception hierarchy is looked through to find if the mutant exception type is super class of the original type, since the base type of exception can also accommodate all its subclass types. When the super class is chosen, it does not satisfy the necessary condition of an effective mutation. These mutants are considered as equivalent mutants and will be exempted.

3.2. CBI - Catch Block Insertion

```java
public void methodFirst(String filename) throws IllegalArgumentException, FileNotFoundException, IOException {
...
try{
in= new FileInputStream(filename);
methodSecond(in);
//methodSecond may throw
//IllegalArgumentException or IOException
} Original:
catch(IOException ioe){... ...}
Mutant(add all the derived types):
\[\Delta\]
catch(IllegalArgumentException iae){... ...}
\[\Delta\]
catchFileNotFoundException fnfe){... ...}
\[\Delta\]
catch(IOException ioe){... ...}
...
}
```

We apply this mutation operator to attach all the derived types before original catch block. This operator enhances the ability of original catch block. JUnit test case needs that tested method exposes the potential exceptions so that JUnit test case can catch the exception. CBI tries to build a complete catch block and hide all exception in the program. However JUnit test case may pass, it proves that test case itself has introduced some unexpected exceptions.

We also analyze the framework of semantic hierarchy. When the original catch type is actually the super class of all the derived types, the mutant is judged as the equivalent mutant. The mutant has the same ability as the original program.

3.3. CBD - Catch Block Deletion

```java
public void methodFirst(String filename) throws IllegalArgumentException, FileNotFoundException, IOException {
...
try{
in= new FileInputStream(filename);
methodSecond(in);
//methodSecond may throw
//IllegalArgumentException or IOException
} Original:
catch(IOException ioe){... ...}
Mutant(delete the whole catch block):
...
}
```

Obviously, the exception propagation path is affected by catch block. CBD delete all the catch clauses following try block. Two test cases which bring up the same exception meet the reachable condition to reach the try point. Deleting the whole catch block will generate some distinctions between the test cases. The operation also will not produce equivalent mutants because CBD just alters one statement point.

We design this mutation operator to aim at two design cases:

First, when one exception is thrown, the current context can not handle the exception appropriately. At this time, the exception is often propagated up to the invoker. The invoker deals with the exception if invoker collects enough information. CBD delete the catch block to let the exception propagate. The test case turns from failed to pass because the exception has been exposed.

Second, the JUnit test case for exception failed sometimes because all the exceptions are concealed by the original program. The quality of this test case has not been proven. CBD deletes the catch block, it is aware that which type of exceptions has been thrown and which test case should pass.

CBD affirms that exceptions are thrown to the upper level and some misapprehend test case are evaluated again.

3.4. PTL - Placing Try Block Later

```java
public void methodFirst(String filename) throws IllegalArgumentException,
```
FileNotFoundException,
IOException
{
… …
byte[] readBytes = null;
Original:
try{
in = new FileInputStream(filename);
}
catch(FileNotFoundException fnfe){… …}
finally{… …}
in.read(readBytes,0,100);
Mutant(add the reference into try block):
try{
in= new FileInputStream(filename);
Δ
in.read(readBytes,0,100);
}
catch(FileNotFoundException fnfe){… …}
finally{… …}
… …
}

Some local variables are used to store the outcome of the method executed. The local variable will be referenced in the following statements. Usually, the method called returned value null but not an exception because the method itself has handled the exception. Any statements which reference the null variable may throw a null pointer exception [21]. It demands that all the linked statements are involved in the try block so that the mutant generates some exceptions missed before.

PTL scans the statements which follow the catch block to grab the references of variables which has been placed in the try block. Because some statements which may throw other types of exceptions. It causes that the program makes different behaviors. Some test cases will fail when they face to new kinds of exceptions.

3.5. CRE – Catch and Rethrow Exception

……

Original:
catch(FileNotFoundException fnfe){… …}
Mutant(re-throw the exception caught):
catch(FileNotFoundException fnfe)
{
Δ fnfe.printStackTrace();
Δ throw fnfe;
}
… …

When a program needs to be recovered from current exception, it executes a rollback. Catch block handles the exception and re-raise the exception. The handler further down the stack can catch this exception. The next handler log the exception and print an appropriately message to the user. Program propagates and handles the exception at the same time. The mutation operator affects two level hierarchies. A test case which can effectively test the upper level will kill the mutant.

The mutation operators above can screen out high quality test cases which can raise most types of exceptions. We have provided a methodology for CBR and CBI to distinguish the equivalent mutant generated through semantic exception hierarchy. Other every mutation operator only makes one mutant at one time. In general, the effort spent on identifying equivalent mutants is acceptable.

4. Result and discussion

This section presents simulated results of mutation operators in Section 3. Firstly we use and evaluate the mutation operators provided by an open source program, and then we apply our mutation operators and evaluate the test cases.

Jumble 1 is a context-free mutation tool. Jumble needs no compilation so that it has high speed of execution. Jumble provides mutation operators for condition, arithmetic, constants return value and switch statements. Jumble takes the quantity of mutants killed as the score of test case. However, these mutation operators are not suitable for exception handler.

Test cases for testing exceptions always get a low score. We take testing method dispense in vending machine program for an example [17]. We design 2 test cases, which can raise one kind of exception separately. Jumble finds 19 mutation points in the method dispense with built-in mutation operators. Jumble scores the first one with 5% and the second one with 15%. The second test case gets a higher score because the exception to be tested is surrounded by more arithmetic conditions so that more built-in mutation operators can be applied. For the method which only has exception handling construct, Jumble takes the test case as a broken test class. So Jumble built-in mutation operators can not distinguish which test case can cover more exceptions.

We then test the whole exception mechanism of vending machine program [20]. It has 7 kinds of exceptions, 3 semantic levels and 4 invoking levels. The program includes 5 methods. We add some outside exception statements to some methods.

In Table 1, the upper table shows the basic information of the five methods of the program. The table below shows the mutants statistic applying different mutation operators.

1 http://jumble.sourceforge.net/
Table 1. Vending machine program applies mutation operators

<table>
<thead>
<tr>
<th>method</th>
<th>try blocks</th>
<th>catch blocks</th>
<th>outside exceptions</th>
<th>invoke method</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>insert</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>returnCoin</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vend</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>dispense</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. Scores of four test sets for method main

<table>
<thead>
<tr>
<th>Test set</th>
<th>CBR</th>
<th>CBI</th>
<th>CBD</th>
<th>CRE</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3/5</td>
<td>0/1</td>
<td>1/2</td>
<td>1/4</td>
<td>5/12</td>
</tr>
<tr>
<td>2</td>
<td>4/5</td>
<td>0/1</td>
<td>1/2</td>
<td>1/4</td>
<td>6/12</td>
</tr>
<tr>
<td>3</td>
<td>4/5</td>
<td>0/1</td>
<td>2/2</td>
<td>2/4</td>
<td>8/12</td>
</tr>
<tr>
<td>4</td>
<td>5/5</td>
<td>0/1</td>
<td>2/2</td>
<td>4/4</td>
<td>11/12</td>
</tr>
</tbody>
</table>

For a method which has a complete exception handler, the mutation operators successfully generate mutants because our mutation operators need to effect on try or catch blocks. For example, the method main generate 5 mutants from CBR and 2 mutants from CBD. If there are only explicit throw statements in the methods, the methods do not need to be mutated. Actually, these methods only provide an option to raise exceptions. Except for CBR and CBI, other mutation operators do not generate any equivalent mutants.

In conclusion, our method does not analyze the control flow of exception handling construct. We form the hierarchy of exceptions and apply mutation operators. These mutation operators produce effective mutants to evaluate test cases effectively. According to the mutation operator set and program analysis, we need a method which can guide to generate test cases for exception handler automatically. The mutation operators are designed based on the characteristic of Java. One direction of future work is that we can find a more general mutation strategy for other advanced languages.

6. References


