CoderChecker

Pratyusha Prathikantham
Software Engineering
Concordia University
Montreal QC Canada
pratyusharaj6@gmail.com

Yu Luo
Software Engineering
Concordia University
Montreal QC Canada
luoyu0811@gmail.com

Zhu Liu
Software Engineering
Concordia University
Montreal QC Canada
lzlurker@gmail.com

ABSTRACT

Static analysis tools can help developer detect potential bugs without running the code. (e.g. memory leak, useless conditions, etc.)

In this paper, we will discuss our design of a static analysis tool which include 10 bug patterns to detect bugs in source code and compare some of the result with a widely used static analysis tool – Spotbugs.

Categories and Subject Descriptors

D.2.9 [Software Engineering]:
Management—Software Quality Assurance (SQA)

KEYWORDS

Static Analysis, Spotbugs, Bug Patterns

1 Introduction

By detecting bug patterns, static analysis is able to examine the code without executing it. The approach greatly helps developers to ensure the standard of the code.

CoderChecker is a tool that is designed to check 10 typical bug patterns for learning purpose. We have also designed some test cases to inspect our design and compare the result with Spotbugs.

2 Detection approach

CoderChecker will loop each .java file in the project and check it with the 10 bug patterns, if any bugs that match the patterns, some error messages will be showed and error will be recorded. Following is the detection approach of each pattern.

1) Class defines equals() but not hashCode()

Method check() in class HashChecker will collect all method declared in a class, if equals was override but hashCode was not, CoderChecker will consider it a bug.

2) Comparison of String objects using == or !=

Method check() in class StringEqualChecker will searching for all the expression in the class statement and check if it's the compare expression was used on a string class. If so, CoderChecker will consider it a bug.

3) Method may fail to close stream on exception

Method check() in class FileStreamCloseChecker will first searching for the stream classes which are not assign or pass to other methods, then search for stream class that has assigned but not closed in the final block. If such case was found, CoderChecker will consider it a bug.

4) Condition has no effect

Method check() in class IfChecker will check for if and else if statement and get their conditions. If the condition is always true/false or a const bool var, CoderChecker will consider it a bug.

5) Inadequate logging information in catch blocks

Method check() in class LogInfoChecker will use a hashmap to store each catch block link to a try. Then loop every item of the hash map. If it has more than 1 item, that means this try has over 1 catch, compare them with each other and search for the same string info. When same string info appearance, CoderChecker will consider it a bug.

6) Unneeded computation in loops

Method check() in class LoopChecker will check for statements and while statements, first collect all return variable name, then recursion for all method call statement. In the loop statements, if there is no variable name appearances which means the method call is meaningless, CoderChecker will consider it a bug.

7) Unused methods

Method check() in class UnusedChecker will first collect all method that has been declared, then collect method call which use field this or empty, compare these two collections, if it contains different methods, it means the methods have never been called. CoderChecker will consider it a bug.

8) Empty exception

Method check() in class EmptyExceptionChecker will check whether the catch statement is empty (except for Comment), if so, CoderChecker will consider it a bug.

9) Unfinished exception handling code

Method check() in class UnfinishedExceptionChecker will check for the catch statement content, if it contains TODO OR FIXME, CoderChecker will consider it a bug.

10) Over-catching an exception with system-termination

Method check() in class OverCatchExceptionChecker will first find those over catch statement (i.e. Exception or RuntimeException). Then loop for those children node, if it has method call named exit or abort CoderChecker will consider it a bug.

3 Design of the test cases

To check the basic performance of CoderChecker, we design following test cases for the bug patterns.

1) Class defines equals() but not hashCode()

Override equals() in class Test, but not override hashCode():

```
 public boolean equals(Object obj) {
       return super.equals(obj);
   }
```

2) Comparison of String objects using == or !=

Use “==”/”!=” to compare String instead of equals():

```
       String a="a";
       String b="b";
       System.out.println(a==b);
       System.out.println(a!=b);
```

3) Method may fail to close stream on exception

"Not-Exist-file.java" is not exist, *.close() won’t be executed at try block, and fileInputStream.close() is not closed at finally block.

```
       FileInputStream fileInputStream =null;
       FileInputStream errorFileInputStream =null;
       try {
```

```
      fileInputStream = new
FileInputStream("Test.java");
       errorFileInputStream = new
FileInputStream("Not-Exist-file.java");
       fileInputStream.close();
       errorFileInputStream.close();
    }catch (Exception e){
       e.printStackTrace();
    } finally {
       errorFileInputStream.close();
    }
```

4) Condition has no effect

Conditions are always true.

```
    if(true){
       System.out.println(ft.format(dNow));
    }
    boolean flag=true;
    if (flag){
       System.out.println(ft.format(dNow));
    }
```

5) Inadequate logging information in catch blocks

Log error information is useless.

```
    try {
       fileInputStream=new
FileInputStream("Test.java");
    } catch (RuntimeException e) {
       Logger.getLogger("sssss");
    } catch (IOException e) {
       Logger.getLogger("sssss");
    }
```

6) Unneeded computation in loops

Unneeded computation because i always larger than 0.

```
    for (int i = 0; i < 100; i++){
       Math.abs(i);
       System.out.println(i);
    }
```

7) Unused methods

Function is not called anywhere.

```
 public void unusedMethod(){
    System.out.println("unused method");
 }
```

8) Empty exception

```
    try {
       fileInputStream=new
FileInputStream("Test.java");
    }catch (Exception e) {
       /*
       * sadfas
       * */
    }
```

9) Unnished exception handling code

Catch block is not finished

```
    try {
       fileInputStream=new
FileInputStream("Test.java");
    }catch (Exception e) {
    // TODO: something
    // FIXME
    }
```

10) Over-catching an exception with system-termination

```
    try {
       fileInputStream=new
FileInputStream("Test.java");
    }catch (Exception e) {
       e.printStackTrace();
       System.exit(0);
    }
```

4 Detection result and discussion

4.1 Result of running CoderChecker on Hadoop 3.0.0 and CloudStack 4.9

Table 1 CoderChecker checking Hadoop&CloudStack shows the result of running CoderChecker on these two projects. We will discuss the result in section 4.3

Bug Pattern Num	Hadoop	CloudStack
1	8	4
2	10	5
3	330	53
4	376	345
5	1	0
6	3689	647
7	10257	5637
8	1644	89
9	25	99
10	13	8
Total	16353	6887

Table 1 CoderChecker checking Hadoop&CloudStack

4.2 Result comparison with Spotbugs

4.2.1 Introduction of Spotbugs

SpotBugs is the spiritual successor of FindBugs which uses static analysis to look for bugs in Java code. SpotBugs checks more than 400 bugs and catigorize them into 10 large classifications. The 4 patterns we are going to discuss belongs to 3 classification respectively.

1. Bad practice (BAD_PRACTICE)
Violations of recommended and essential coding practice. Examples include hashcode and equals problems, cloneable idiom, dropped exceptions, Serializable problems, and misuse of finalize. We strive to make this analysis accurate, although some groups may not care about some of the bad practices.[1]

2. Experimental (EXPERIMENTAL)
Experimental and not fully vetted bug patterns.[1]

3. Dodgy code (STYLE)
Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, switch fall through, unconfirmed casts, and redundant nullcheck of value known to be null. More false positives accepted. In previous versions of SpotBugs, this category was known as Style.[1]

4.2.2 Result of running SpotBugs on Hadoop 3.0.0 and CloudStack 4.9

Table 2 SpotBugs checking Hadoop&CloudStack shows the result of running SpotBugs on these two projects.

Bug Pattern Num	Hadoop	CloudStack
1	0	5
2	2	14
3	2	104
4	9	28
Total	13	151

Table 2 SpotBugs checking Hadoop&CloudStack

Comparing the two tables, we notice that CoderChecker reports way more bugs on hadoop than SpotBugs(724:13). And for CloudStack, the difference is not that obvious. SpotBugs reports more bugs on pattern 1,2,3; CoderChecker reports more on pattern 4. Totally is CoderChecker 407:SpotBugs 151.

It is not necessary that the tool which reports more bugs is better because there could be false positives. To make comparison, we will do some deep research by randomly pick some reported bugs by each tool and check whether it was false positive or ture bug.

4.3 Discussion

	SpotBugs		CoderChecker	
Bug Pattern Num	Hadoop	CloudStack	Hadoop	CloudStack
1	0	5	8	4
2	2	14	10	5
3	2	104	330	53
4	9	28	376	345

Table 3. Comparison for the pattern 1-4

4.3.1 Pattern 1: HE_EQUALS_USE_HASHCODE: Class defines equals() and uses Object.hashCode()

The 8 bugs CoderChecker found for hadoop are come from test cases, e.g. TestGenericWritable.java=>[Error] Line 119. And SpotBugs reported 0 bug. So we have reason to believe SpotBugs ignore those test cases.

The test result for CloudStack between two tools has a high similarity(3 out of 4 bugs reported are the same). e.g. AgentAttache.java=>[Error] Line 333 VS At AgentAttache.java:[lines 336-344].

From the result, we can infer that CoderChecker uses similar algorithm with Spotbugs at pattern 1.

4.3.2 Pattern 2: ES_COMPARING_STRINGS_WITH_EQ: Comparison of String objects using == or !=

For pattern 2, the difference is also caused by SpotBugs didn't check the test cases of hadoop.

For CloudStack however, we notice that SpotBugs not only check String objects, but also check other types of objects. e.g. At TransactionLegacy.java:[line 183].

According to the result, we believe that Spotbugs checks more kinds of objects other then String when compare using == or !=. Our CoderChecker only test on String objects.

4.3.3 Pattern 3: OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE: Method may fail to clean up stream or resource on checked exception

In Hadoop, many resource were cleaned up by some self defined methods e.g. cleanupWithLogger(Logger logger, java.io.Closeable... closeables). CoderChecker fail to detect such way to clean up and reported lots of false positive.
In CloudStack, we found some false positive occured because of a Socket opening without closing. e.g. JuniperSrxResource.java=>[Error] Line 542. CoderChecker will report it as a bug. However, as a server, socket supposed to keep opening.

Meanwhile, CloudStack detected several sql statement clean up failure which CoderChecker didn't considered, e.g. Obligation to clean up resource created at StorageManagerImpl.java:[line 1246] is not discharged.

4.3.4 Pattern 4: UC_USELESS_CONDITION: Condition has no effect

For this pattern, CoderChecker reported lots of bugs that Spotbugs didn't. After analysis our algorithm. We found that in our design we will report bug as long as a condition was predefined true or false. That makes lots of our detections false positive.

5 Conclusion:

In this paper, we used static analysis tool spotbugs to detect the bugs in source code,by implementing the tool in Java. We have used CodeChecker to detect bug patterns and listed the detection approach of each pattern individually.

We then designed the test cases to check the performance of CodeChecker and run the CodeChecker on both Hadoop 3.0.0 and CloudStack 4.9. Then, we run Spotbugs on Hadoop 3.0.0 and CloudStack 4.9. and compared both the results. Thus we have successfully implemented a total of 10 Bug patterns and designed test case for each bug pattern.

After compare the result of pattern 1-4 of our tool and Spotbugs running on Hadoop 3.0.0 and CloudStack 4.9, we found that for pattern 1,2 the result is similar, so we believe our algorithm is working for the two patterns. But for pattern 3,4 we found several design flaw in CodeChecker which result in many false positive reporting.

Also, by observing the rest of our result, we think that CodeChecker still have lots to improve. Due to the time limitation, we can only deliver it with basic fulfillment of the requirements.

6 Future work and Github link

We will try to fix these defects and optimize our algorithm for CodeChecker.

Following is the link to Github:
https://github.com/DavidLuo06/CoderChecker.git

Link to our documentation:
https://drive.google.com/open?id=1ONdVDQ0o08ENZDROWlmcb1RiCTRyt1oc

We will also put all of our analysis raw data on Github.

REFERENCES

[1] https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html