A Static Analysis Tool for detecting bug patterns

Urbashi Bhattacharjee
Concordia University
Urbashi0931@gmail.com

Dr. Jinque Yang
Assistant Professor
Concordia University

ABSTRACT

Static Analysis Tools are used for better detection of code. In this paper, A tool is made for detecting three bug patterns from a source code. This work can be divided into two phases , i) requirement analysis and ii) Detection. The metric used for evaluating performance of this tool is False Positive rates of Bugs Detected. This tool has been run on large scale open source system cloudstack. However, this approach has made better false positive rates inadequate logging information bugs at only 10%. The detection works better for Empty control flow bug pattern that is 109 in total.

Keywords

Software Development Lifecycle; Static Analysis ; testing.

1. INTRODUCTION

Static Analysis tool or analyzer can analyze a source code without having run time information of the source code. For many years static analysis have been used in Software Development Life Cycle as a basic testing approach against requirements. Requirement Analysis is undoubtedly very significant in any software engineering projects, because they lead to making right software for meeting business needs. Findbugs is one of the popular static Analysis tools used for software testing. But there is a constant need for inventing new tool for accommodating with updates about bug patterns and organization need.

In this paper, a basic Java Application is built for detecting three bug patterns in java. Java Parser is used for successful parsing the java source code containing at least one or no error at initial phase. Later the analyzer was run on an open source project named CloudStack [4,9].

2. REQUIREMENT ANALYSIS

This paper focuses on implementing three bug patterns for successful static analysis of the source code. Therefore in requirement analysis phase understanding the bug pattern requirements is important.

RQ1: Can the static analysis tool detect only hashcode() but not equals() implementation of a given source code?

RQ2: Can this tool detect one parameterized duplicate logging error message in different catch() blocks of the same try() block?

RQ3: Can this static analysis tool detect empty control flow as a bug for a given source code?

3. DETECTION APPROACH

3.1 Class defines Hashcode() but not equals()

In this bug pattern, Static analyzer looks for only hashcode() implementation as a bug, because it does not have equals() with it. In this approach, first of all the parser looks for Method Declarations in a given source code. Secondly, it matches the name of the method to be either “equals” or “hashcode” . Type and parameter size of these methods are also checked for better detection. For the sake of this paper, this bug pattern is abbreviated as HE.

3.2 Inadequate Logging information in Catch blocks

In this approach parser looks for the catch clause and and gets each statement as a child node. It analyzes each statements that contains logger/log.* format. If it gets duplicate information on the catch block with one parameter it returns a bug to the tester. . For the sake of this paper, this bug pattern is abbreviated as IL.

```
Try{
}
Catch (Exception e)
{
Log.warn(“X”);
}
Catch (Exception e)
{Log.warn(“X”);
}}
```

Figure : Sample bug pattern

3.3 Empty Control Flow

In analyzing empty control block, the application analyses switch, if, else-if, if…else, while, do…while, while blocks. In Javaparser, SwitchEntryStatement class defines all the information about how switch statements are analyzed. Therefore it looks for empty cases in the switch block and returns an error of bug found. Other control flow parsing classes looks for Block Statements if available after each keyword declaration in code. If block is empty, then it has to be a bug. . For the sake of this paper, this bug pattern is abbreviated as ECF.

The switch control can be like as follows-

```
switch(test){                    switch(test)
}                                   {
                                    Case 1:
                                     }
   (a)                                (b)
```

Figure 1: (a) and (b) both are empty switch blocks

3.4 Detection Results

Metric used for evaluating detection results is False positive Rate. For better performance the lesser false positive rates means better detection results. It has to be noted that False Positive Rates are analyzed for 20 bugs for each pattern.

Table 1. Table captions should be placed above the table

Bug Pattern	No of True positive(s)	No of False Positives(in 20 bugs analyzed)
HE	0	0
IL	8	2
ECF	109	5

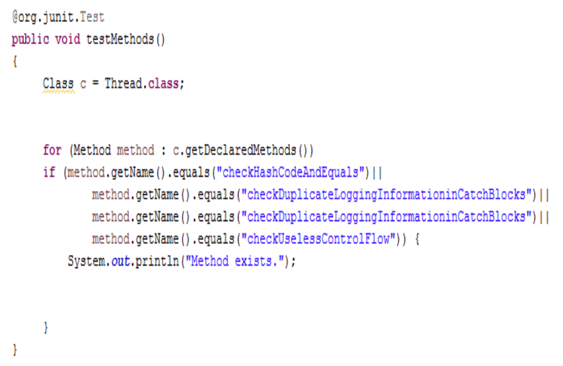
According to the Table 1, False positive rate should be 10% and 25% for IL and ECF. Hence, code quality has to be improved for better detection results.

3.5 Testing our code

Test cases are made for each bug pattern. Our approach uses JUNIT for test cases. In total, it has 5 test cases. The testing file is AppTest.Java in the github repository.

The test cases are as follows-

public void testMethods(): This test case contains that, Main() method has invoked all necessary methods needed for static analysis of system.



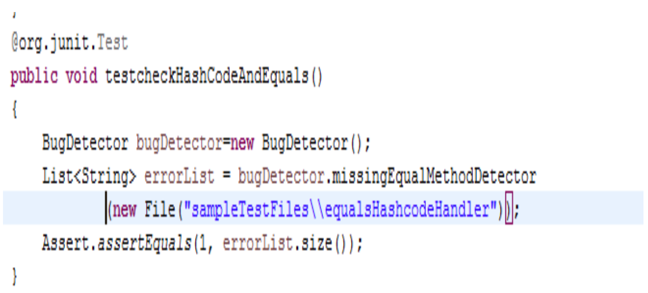
```
@org.junit.Test
public void testMethods()
{
    Class c = Thread.class;


    for (Method method : c.getDeclaredMethods())
    if (method.getName().equals("checkHashCodeAndEquals")||
            method.getName().equals("checkDuplicateLoggingInformationinCatchBlocks")||
            method.getName().equals("checkDuplicateLoggingInformationinCatchBlocks")||
            method.getName().equals("checkUselessControlFlow")) {
        System.out.println("Method exists.");


    }
}
```

Figure 2: Code Snippet of this test

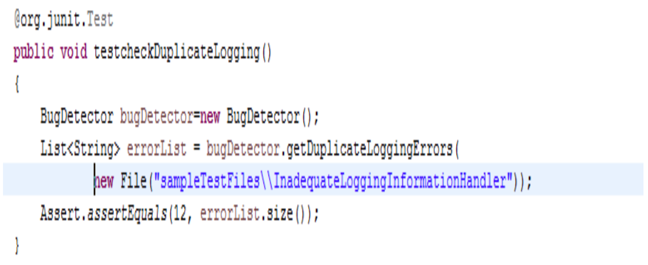
public void testcheckHashCodeAndEquals(): This test takes a file from testFile Folder in the repository and checks if Bug Pattern Returns true.



```
@org.junit.Test
public void testcheckHashCodeAndEquals()
{
    BugDetector bugDetector=new BugDetector();
    List<String> errorList = bugDetector.missingEqualMethodDetector
            (new File("sampleTestFiles\\equalsHashcodeHandler"));
    Assert.assertEquals(1, errorList.size());
}
```

Figure 3: Code snippet of test

public void testcheckDuplicateLogging(): This test takes a file input and checks for Duplicate Logging information. For the sake of this assignment, the test case contained 12 bugs in total. So in the end error size is compared for expected bug received.



```
@org.junit.Test
public void testcheckDuplicateLogging()
{
    BugDetector bugDetector=new BugDetector();
    List<String> errorList = bugDetector.getDuplicateLoggingErrors(
            new File("sampleTestFiles\\InadequateLoggingInformationHandler"));
    Assert.assertEquals(12, errorList.size());
}
```

Figure 4: testcheckDuplicateLogging() test method

3.6 Experimental Setup

For this assignment, I have used Java Eclipse IDE for commiters edition and a basic Maven project is created for easier dependency injection. The version of Junit is 4.12. The version for java parser is 3.6.12. This can be verified from pom.xml in the project directory. Our code is available at https://github.com/urbashi0931/SOEN-7481_Assignment. It has to be noted that, this paper has used external github library named tomassoti.support.

4. Future Work

In future, this assignment has to be analyzed for better code quality and better detection approach. This paper work, does not have test cases for control flow statement. However, it can be elaborated for other bug patterns too.

5. ACKNOWLEDGMENTS

My sincere thanks goes to Dr. Jinque Yang and TA's for help.

6. REFERENCES

[1] "Eclipse," http://www.eclipse.org/.

[2] "FindBugs," http://findbugs.sourceforge.net.

[3] "Java Parser" https://tomassetti.me/getting-started-with-javaparser-analyzing-java-code-programmatically